



北京华章图文信息有限公司

国外经典教材



Classical Texts From Top Universities

C++ 程序设计语言 (特别版)

*The C++ Programming
Language*
(Special Edition)

(美) Bjarne Stroustrup 著
贝尔实验室
裘宗燕 译



机械工业出版社
China Machine Press



Pearson Education
培生教育出版集团

本书由C++语言的设计者编写,是有关C++语言的最全面、最权威的著作。本书覆盖了标准C++以及由C++所支持的关键性编程技术和设计技术。标准C++较以前的版本功能更强大,其中许多新的语言特性,如名字空间、异常、模板、运行时类型声明等使得新技术得以直接应用。本书围绕语言及库功能来组织,内容涉及C++的主要特征及标准库,并通过系统软件领域中的实例解释说明一些关键性的概念与技术。

本书的目的是帮助读者深刻地理解C++如何支持编程技术,从而成为一名优秀的编程人员 and 设计人员。本书适合作为高校计算机专业C++语言和面向对象编程等课程的教科书,也是C++程序员和爱好者必备的参考书。

Simplified Chinese edition Copyright © 2002 by PEARSON EDUCATION NORTH ASIA LTD and China Machine Press.

Original English language title: The C++ Programming Language, Special Edition by Bjarne Stroustrup, Copyright © 2000.

All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison Wesley Longman, Inc.

This edition is authorized for sale only in People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书封面贴有Pearson Education培生教育出版集团激光防伪标签,无标签者不得销售。版权所有,侵权必究。

本书版权登记号:图字:01-2001-2195

图书在版编目(CIP)数据

C++程序设计语言(特别版)/(美)斯特朗斯特鲁普(Stroustrup, B.)著;裘宗燕译.-北京:机械工业出版社,2002.7

(国外经典教材)

书名原文:The C++ Programming Language, Special Edition

ISBN 7-111-10202-9

I. C… II. ①斯…②裘… III. C语言-程序设计-教材 IV. TP312

中国版本图书馆CIP数据核字(2002)第029325号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:温丹丹 杨海玲

北京牛山世兴印刷厂印刷·新华书店北京发行所发行

2002年7月第1版第1次印刷

787mm×1092mm 1/16·58.75印张

印数:0 001-8 000册

定价:85.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

“国外经典教材”是响应教育部提出的使用外版教材的号召，为国内高校的计算机本科教学度身订造的。在广泛地征求并听取丛书的“专家指导委员会”的意见后，我们最终选定了这20多种篇幅内容适度、讲解鞭辟入里的教材，其中的大部分已经被M.I.T.、Stanford、U.C. Berkley、C.M.U.等世界名牌大学采用。丛书不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

电子邮件：hzedu@hzbook.com

联系电话：(010) 68995265

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元
石教英
张立昂
邵维忠
周克定
郑国梁
高传善
裘宗燕

王 珊
吕 建
李伟琴
陆丽娜
周傲英
施伯乐
梅 宏
戴 葵

冯博琴
孙玉芳
李师贤
陆鑫达
孟小峰
钟玉琢
程 旭

史忠植
吴世忠
李建中
陈向群
岳丽华
唐世渭
程时端

史美林
吴时霖
杨冬青
周伯生
范 明
袁崇义
谢希仁

Preface to the Chinese Edition

This book is the most complete and up-to-date book on Standard C++. It is also the most widely read and most widely available. By my current best count, it is now available in 17 languages (see <http://www.research.att.com/~bs/covers.html>). This translation is based on a text that has been improved by thousands of suggestions from readers.

I am particularly pleased that this book is now more easily accessible to Chinese programmers and students. Chinese colleagues have pointed out the need for a translation to me, as has many Chinese programmers in email. Not being a native English speaker myself, I appreciate the need - and anyway, I just love my collection of translations as a graphic example of the widespread use of C++.

Naturally, some people prefer the original English text whereas others feel that a translation into their native language removes a barrier to understanding. I have known programmers who used both the original version and a translation to have the advantage of using their native language and also of having English available to communicate with programmers worldwide.

This book covers Standard C++, its standard library, and the fundamental programming techniques supported by C++, such as Object-Oriented Programming and Generic Programming. The aim is not just to explain the language facilities, but to provide sufficient information of their effective use for a programmer to cope with major projects. For that discussion of design is essential.

In August 1998 the ISO standard for C++ (ISO/IEC 14882 *Standard for the C++ Programming Language*) was ratified (by a 22-0 vote of national standards committees). This was a milestone for C++ inaugurating a new era of stability and advances in tools and techniques.

For me, the bottom line is that Standard C++ is a better approximation to my aims for C++ than any previous version. Standard C++ and its standard library allows me to write better, more elegant, and more efficient C++ programs than I have been able to in the past.

The aim of the standards effort was to specify a language and a library that will serve every part of the user community well without favoring any particular group of users, any particular company, or any particular country over others. It was an open and fair process aimed at quality and consensus.

One potential problem with an open and democratic standards process is "design by committee." This was largely avoided for C++. One reason was that I served as the chairman of the working group for language extensions. In that capacity, I evaluated all suggestions for major extensions and wrote the final version of the ones I, the working group, and the committee deemed both worthwhile and feasible. Thus, the primary activity of the committee was discussion of relatively complete designs presented to it, rather than design itself. Similarly, the major new part of the standard library - the "STL" providing a general, efficient, type-safe, and extensible framework of containers, iterators, and algorithms was primarily the work of one man: Alexander Stepanov.

Importantly, the C++ standard is not just a document. It is embodied in C++ implementations. All the major implementations now implement the standard with very minor exceptions. To help keep vendors honest, at least two companies now offer validation suites for Standard C++. Thus, the code I write now use — where appropriate — most of the facilities offered by Standard C++ and described in this edition of *The C++ Programming Language*.

The improvements to C++ language and the addition of a standard library have made a significant difference in the way I am able to write my code. My programs are now shorter, cleaner, and more efficient than they used to be. This is a direct result of Standard C++'s better, more systematic, and cleaner support for abstraction. Better support for facilities such as templates and exceptions reduce the need to deal with lower-level and messier facilities. Also, the last few years have seen the emergence of many new design and programming techniques, which are reflected in the presentation approach and examples of this book.

C++ can now be taught as a higher-level language. That is, the initial emphasis can be on algorithms and containers, rather than fiddling with bits, unions, C-style strings, arrays, etc. Naturally, lower-level concepts (such as arrays, non-trivial uses of pointers, and casts) must eventually be taught/learned. However, the presentation of such facilities can be postponed until a novice C++ programmer, reader, or student has gained the maturity to see them in the context of the higher-level concepts that they are used to implement.

In particular, I cannot overemphasize the importance of the statically type-safe strings and containers over programming styles involving lots of macros, casts, and arrays. In *The C++ Programming Language* I have been able to eliminate essentially all uses of macros and to reduce the use of casts to the handful of cases where they are essential. I consider the C/C++ form of macros a serious deficiency — which now has been made largely redundant by proper language facilities such as templates, namespaces, inline functions, and constants. Similarly, extensive use of casting is — in any language — a sign of weak design. Both macros and casts are major sources of errors. Being able to do without them makes C++ programming much safer and more elegant.

Standard C++ is meant to change the way we program in C++, to change the way we design our programs, and to change the way we teach C++ programming. Such changes do not happen overnight. I encourage you to take a long hard look at Standard C++, at the design and programming techniques used in *The C++ Programming Language*, and the way you program. I suggest that major improvements are possible. Do keep a cool head, though. There are no miracles, and using a language facility or a technique that you only partially understand in production code is dangerous. Now is the time to explore and experiment — the really major benefits of Standard C++ you reap only through the understanding of new concepts and new techniques.

Enjoy!

Bjarne Stroustrup

中文版序

本书是讲述标准C++的最完整和最新的著作，它拥有最多的读者，使用也最为广泛。按我目前的统计，本书已经被翻译成17种语言（参见<http://www.research.att.com/~bs/covers.html>）。所以，这个译本所依据的原文，已经从成千上万的读者建议中获益匪浅。

现在，中国的程序员和莘莘学子能够更容易地读到本书，对此我尤感欣慰。我的中国同事，还有许许多多中国的程序员（通过电子邮件）早就向我建议有必要将本书译为中文。因为自己的母语也不是英语，我当然也认识到了这种必要性——何况，我还非常喜欢拿本书译本的总数作为C++得到广泛应用的活生生的例子。

自然了，所谓“仁者乐山，智者乐水”，有人会更喜欢英文原版，而另一些人则会感觉阅读翻译成母语的版本更能消除理解上的障碍。我认识许多程序员同时使用原版和译本，这样既能发挥母语的优势，又能用英语与全世界的程序员进行交流。

本书涵盖了标准C++、它的标准库和C++所支持的基本技术，如面向对象程序设计和通用型程序设计。其目的不仅仅是阐述语言的功能，还要提供如何行之有效地使用这些功能的信息，使程序员足以应付大多数开发项目。因此其中对设计的讨论非常重要。

1998年，ISO的C++标准（ISO/IEC 14882 *Standard for the C++ Programming Language*）得到了批准（各国标准委员会以22-0全票通过）。这是C++发展史上的一个里程碑，开创了C++工具和技术稳定发展的新纪元。

对我本人而言，其中关键在于，标准C++相对于以前的任何版本，更接近于我对C++的目标。标准C++及其标准库使我能够编写出比过去更好、更优雅、更高效的C++程序。

标准化的目的是为一种语言和一个库制定规范，使其能够服务于所有用户群体，而不至偏向于某个用户群、某个公司或某个国家。这是一个以保证质量和达成共识为目的的开放、公正的过程。

开放和民主的标准化过程存在一个潜在的问题：所谓“由委员会设计”。这在C++的标准化中基本上被避免了。原因之一在于，我担任了语言扩展工作组的主席。在此位置上，我负责评估所有关于主要语言扩展方面的建议，并就那些我本人、工作组和委员会都认为值得和可行的建议撰写最终版本。因此，委员会的主要活动是讨论提交上来的相对完整的设计，而不是自己来设计。与此类似，标准库的主要新增部分——“STL”（为容器、迭代器和算法提供了通用的、高效的、类型安全的和可扩展的框架），主要都源自一个人——Alexander Stepanov的工作成果。

重要的是，C++标准不仅仅是一份文档。它已经在各种C++实现产品中得到了体现。所有主要的C++实现产品现在都实现了标准，只有极少的几个例外。为了帮助厂商更好地实现标准，现在至少有两个公司提供了标准C++的验证套件。因此，我现在写代码，只要合适，都会用到标准C++提供的和本书这一版中讲述的功能。

C++语言的改进和标准库的增加，使我自己编写代码的方式发生了显著变化。现在我的程序比原来更加简洁、更加高效。这直接得益于标准C++对抽象更好、更系统和更纯粹的支持。

对模板和异常等功能更好的支持，使对底层处理和更混乱的功能的需要大大降低了。而且，最近几年出现了许多新的设计和编程技术，这在本书的表达方法和实例中都有所反映。

C++现在可以作为高级语言来讲授了。也就是说，重点一开始就可以放在算法和容器上，而不用再在什么位呀、联合呀，C风格字符串，数组等等东西上纠缠不清了。自然，底层的概念（如数组、重要的指针应用和强制转换）最终还是要教要学的。但是，可以等到作为新手的C++程序员、读者或学生已经成熟，能够在实现这些功能的高级概念的大背景中看待它们的时候，再对这些功能进行阐释。

我想特别强调（怎么强调都不过分）的是，应该多使用静态类型安全的字符串和容器，而不要学那些使用大量宏、强制转换和数组的编程风格。在本书中，我能够根本就不用宏，并且只在很少的非用不可的情况下才使用强制转换。我认为C/C++形式的宏是一种严重的缺陷——现在因为有了模板、名字空间、内联函数和常量这些正确的语言功能，它很大程度上更是一种多余了。同样，在任何语言中，强制转换的大量使用都是设计不良的标志。宏和强制转换是错误的主要渊藪。不用它们也能工作，这一点大大提高了C++编程的安全性和优雅性。

标准C++改变了我们使用C++编程、设计程序以及教授C++编程的方式。这些变化不可能“毕其功于一役”。我鼓励你在标准C++、在本书中所用的设计和编程技术，以及自己的编程方式上好好下一番功夫。我想脱胎换骨是有可能的。但是别太死心眼了。奇迹是不存在的，在产品代码中使用仅仅一知半解的语言功能和技术是相当危险的。现在该开始探索，开始试验了——标准C++真正对你有所裨益的地方，就在理解新概念和新技术的旅程中！

旅途愉快！

Bjarne Stroustrup

译者序

Bjarne Stroustrup的《*The C++ Programming Language*》是有关C++语言的第一部著作。毫无疑问，它是关于C++语言及其程序设计的最重要著作，在此领域中的地位是无可替代的。《*The C++ Programming Language*》一书伴随着C++语言的发展演化而不断进步，经过第1版（1985年）、第2版（1991年）、第3版（1998年），本书的英文原书是《*The C++ Programming Language*》第3版经过补充和修订后的“特别版（2000）”（对应于国内引进的影印本）。对于这个中译本，我想说的第一句话就是“来得太晚了”。

要学习C++语言和程序设计，要将C++应用于程序设计实践，本书自然是必读之书。这个“特别版”以标准化的C++语言为基础，讨论了C++的各种语言特征和有效使用这一语言的程序设计技术。书中也用了大量的篇幅，在标准库以及一般软件开发的环境下，讨论了使用C++语言编程和组织程序的许多高级技术。本书内容覆盖了C++语言及其程序设计的各个方面，其技术深度与广度是举世公认的。

然而，作者讨论的并不仅是C++语言及其程序设计。本书的讨论远远超出这一范围，第四部分用了大量的篇幅去讨论软件开发过程及其问题。即使是在介绍C++语言及其程序设计的具体问题时，作者也常在程序结构、设计和软件开发的大环境下，提出自己的许多认识。作者有很强的计算机科学与技术基础，在系统软件开发方面极富经验，他所提出的观点和意见值得每个在这个领域中工作的人的重视。

当然，重视并不是盲从。在Stroustrup的两本关于C++的重要著作（本书和《C++语言的设计与演化》（已由机械工业出版社出版））中，都有这样一句话使我印象深刻：希望读者带着一种健康的怀疑态度。看来这是作者深深铭刻在心的一种思想，也特别值得国内每个从事信息技术，或者努力向这个方向发展的人注意。从来就没有什么救世主，Stroustrup不是在传道，他只是在总结和论述自己在这个领域中工作的经验。请不要将本书中的东西作为教条，那也一定是本书作者所深恶痛绝的。

许多人说本书比较难读，这种说法有一定道理。真正理解本书的一般性内容需要花一些时间，融会贯通则更需要下功夫。理解本书的内容不仅需要去读它，还需要去实践。问题是，花这个时间值吗？作者在讨论C++语言的设计时提出了一个观点：你从C++语言中的收获大致与在学习实践这个语言的过程中所付出的努力成正比；而且C++是一个可以伴随你成长的语言。同样的话也适用于本书。如果你致力于将自己发展成一个职业的程序员，或者要在计算机方面的技术领域中长期工作下去，我认为，你从本书中的收获大致也会与你所花的时间成正比，这本书也是一本能够伴随你成长的书。

当然，这本书也不是没有缺陷的。由于作者有着极其丰富的实践经验，因此，当他想要论述一个问题、提出一个观点时，常会想到在自己长期实践中最适合说明这个问题的示例，用几句简短的话引述有关的情况。由于作者对C++谙练有加，因此，在讨论中有时会不知不觉地将某些并不显然的东西当做不言自明的事情提出来。而对于许多初学者而言，这些都可能成为学习中的障碍。为了帮助这部分读者，我也在书中一些地方加入了少量注释，解释一

些背景性情况。过多过滥的注释会增大书的篇幅，干扰读者阅读，绝不会是大家都喜欢的方式。因此我在这样做的时候也很有节制，希望不会引起读者的反感。

由于读者的水平有极大差异，对一些人很熟的东西，对另一些人可能就会莫名其妙。我无法解决所有问题，但也希望能为广大读者做一点服务性的工作。为了帮助初学者入门，为使本书（包括其中文译本）能更好地在国内计算机领域中发挥作用，也为了关心本书、学习本书的人们能够有一个交流经验、传播认识的场所，我将在下面地址维护一个有关本书的信息、情况、认识和意见的网页：

<http://www.math.pku.edu.cn/teachers/qiuzy/cpp.htm>

在其中收集有关的信息，记录朋友们（包括我自己）的认识与意见，提出的问题和相应的认识，有关这一译本的勘误信息，原英文书的更正与更新信息等。欢迎读者提供自己的见解和问题，提供有价值的线索。我没时间去创建与维护一个“纯的”C++语言及其程序设计的讨论组（确实需要这样的场所，而有关VC等的讨论组倒是太多了。如果有人愿做，我乐得坐享其成、积极参与并尽可能提供帮助），只想抽空将接收到的和自己写的东西编辑公布。与我联系可以发email: qzy@math.pku.edu.cn。我还将把有关《C++语言的设计与演化》一书的相关信息也放在那里，供大家参考。

还请读者注意，本书的英文原版书是“特别版”的第1次印刷，即“第3版”的第11次印刷，也是目前国内可买到的影印本的原书。在那以后，作者在重印时不断更正书中的错误，并修改了少量的程序示例。最新的重印是第16次印刷，有关情况可从作者的网页或上面网址找到。由于一些情况，本书无法按最新的重印本翻译，但我还是参考了作者的网页，在译文中尽可能地采纳了有关勘误信息。此外，在翻译过程中我也发现了一些错误。经与作者通过电子邮件讨论取得了一致意见，有关更正反映在本书里。由于这些原因，本书在个别地方的说法可能与读者手头的英文原书有异。如果想确认有关情况，请查看原书的勘误信息。

裘宗燕

2002年2月于北京大学数学学院信息科学系

译者简介



裘宗燕，北京大学数学学院信息科学系教授。长期从事计算机软件与理论、程序设计语言和符号计算方面的研究和教学工作。已出版多部著作和译著，包括：《程序设计语言基础》（译著，北京大学出版社，1990），《Mathematica数学软件系统的应用与程序设计》（编著，北京大学出版社，1994），《计算概论（上）》（合著，高等教育出版社，1997），《从问题到程序——程序设计与C语言引论》（编著，北京大学出版社，1999），《程序设计实践》（译著，机械工业出版社，2000），《C++语言的设计和演化》（译著，机械工业出版社，2002），《程序设计语言——概念和结构》（合译，机械工业出版社，2002）等。

序

去编程就是去理解。

——Kristen Nyggard

我觉得用C++ 编程序比以往更令人感到愉快。在过去这些年里，C++ 在支持设计和编程方面取得了令人振奋的进步，针对其使用的大量新技术已经被开发出来了。然而，C++ 并不就是好玩。普通的实际程序员在几乎所有种类和规模的开发项目上，在生产率、可维护性、灵活性和质量方面都取得了显著的进步。到今天为止，C++ 已经实现了我当初对它的期望中的绝大部分，还在许多我原来根本没有梦想过的工作中取得了成功。

本书介绍的是标准C++^①以及由C++ 所支持的关键性编程技术和设计技术。与本书第1版所介绍的那个C++ 版本相比，标准C++ 是一个经过了更仔细推敲的更强大的语言。各种新的语言特征，如名字空间、异常、模板，以及运行时类型识别，使人能以比过去更直接的方式使用许多技术，标准库使程序员能够从比基本语言高得多的层面上起步。

本书第2版中大约有三分之一的内容来自第1版。这个第3版则是重写了比例更大的篇幅的结果。它提供的许多东西是大部分有经验的程序员也需要的，与此同时，本书也比它的以前版本更容易供新手入门。C++ 使用的爆炸性增长和作为其结果的海量经验积累使这些成为可能。

一个功能广泛的标准库定义使我能以一种与以前不同的方式介绍C++ 的各种概念。与过去一样，本书对C++ 的介绍与任何特定的实现都没有关系；与过去一样，教材式的各章还是采用“自下而上”的方式，使每种结构都是在定义之后才使用。无论如何，使用一个设计良好的库远比理解其实现细节容易得多。由于这些情况，在假定读者已经理解了标准库的内部工作原理之前，就可以利用它提供许多更实际更有趣的例子。标准库本身也是程序设计实例和设计技术的丰富源泉。

本书将介绍每种主要的C++ 语言特征和这个标准库，它是围绕着语言和库功能组织起来的。当然，各种特征都将在使用它们的环境中介绍。也就是说，这里所关注的是将语言作为一种设计和编程的工具，而不是语言本身。本书将展示那些使C++ 卓有成效的关键性技术，讲述为掌握它们所需要的那些基本概念。除了专门阐释技术细节的那些地方之外，其他示例都取自系统软件领域。另一本与本书配套出版的书《带标注的C++ 语言标准》(*The Annotated C++ Language Standard*)，将给出完整的语言定义，所附标注能使它更容易理解。

本书的基本目标就是帮助读者理解C++ 所提供的功能将如何支持关键性的程序设计技术。这里的目标是使读者能远远超越简单地复制示例并使之能够运行，或者模仿来自其他语言的程序设计风格。只有对隐藏在语言背后的思想有了一个很好的理解之后，才能真正掌握这个语言。如果有一些具体实现的文档的辅助，这里所提供的信息就足以对付具有挑战性的真实世界中的重要项目。我的希望是，本书能帮助读者获得新的洞察力，使他们成为更好的程序员和设计师。

① ISO/IEC 14882, C++程序设计语言标准。

致谢

除了第1版和第2版的致谢中所提到的那些人之外,我还要感谢Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger和Chris Van Wyk,他们对第3版各章的初稿提出了许多评论和意见。没有他们的帮助和建议,这本书一定会更难理解,包含更多的错误,没有这么完全,当然也可能稍微短一点。

我还要感谢C++ 标准化委员会的志愿者们,是他们完成了规模宏大的建设性工作,才使C++ 具有它今天这个样子。要罗列出每个人就会有一点不公平,但一个也不提就更不公平,所以我想特别提出Mike Ball, Dag Brück, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, José Lajoie, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov和Mike Vilot,他们中的每个都在C++及其标准库的某些方面直接与我合作过。

在这本书第一次印刷之后,许多人给我发来电子邮件,提出更正和建议。我已经在原书的结构里响应了他们的建议,使后来出版的版本大为改善。将本书翻译到各种语言的译者也提供了许多澄清性的意见。作为对这些读者的回应,我增加了附录D和附录E。让我借这个机会感谢他们之中特别有帮助的几位: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Dietmar Kühl, Nicolai Josuttis, Nathan Myers, Paul E. Sevinc, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai(Mike) Yang和Dennis Yelle。

Bjarne Stroustrup
Murray Hill, 新泽西

第2版序

前路漫漫。

——Bilbo Baggins

正如在本书的第1版所承诺的，C++ 为满足其用户的需要正在不断地演化。这一演化过程得助于许多有着极大的背景差异，在范围广泛的应用领域中工作的用户们的实际经验的指导。在第1版出版后的六年中，C++ 的用户群体扩大了不只百倍，人们学到了许多东西，发现了许多新技术并通过了实践的检验。这些技术中的一些也在这一版中有所反映。

在过去六年里所完成的许多语言扩展，其基本宗旨就是将C++ 提升成为一种服务于一般性的数据抽象和面向对象程序设计的语言，特别是提升为一个可编写高质量的用户定义类型库的工具。一个“高质量的库”是指这样的库，它以一个或几个方便、安全且高效的类的形式，给用户提供了一个概念。在这个环境中，安全意味着这个类在库的使用者与它的供方之间构成了一个特殊的类型安全的界面；高效意味着与手工写出的C代码相比，这种库的使用不会给用户强加明显的运行时间上或空间上的额外开销。

本书介绍的是完整的C++ 语言。从第1章到第10章是一个教材式的导引，第11章到第13章展现的是一个有关设计和软件开发问题的讨论，最后包含了完整的C++ 参考手册。自然，在原来版本之后新加入的特征和变化已成为这个展示的有机组成部分。这些特征包括：经过精化后的重载解析规则和存储管理功能，以及访问控制机制、类型安全的连接、*const*和*static*成员函数、抽象类、多重继承、模板和异常处理。

C++ 是一个通用的程序设计语言，其核心应用领域是最广泛意义上的系统程序设计。此外，C++ 还被成功地用到许多无法称为系统程序设计的应用领域中。从最摩登的小型计算机到最大的超级计算机上，以及几乎所有操作系统上都有C++ 的实现。因此，本书描述的是C++ 语言本身，并不想试着去解释任何特殊的实现、程序设计环境或者库。

本书中给出的许多类的示例虽然都很有用，但也还是应该归到“玩具”一类。与在完整的精益求精的程序中做解释相比，这里所采用的解说风格能更清晰地呈现那些具有普遍意义的原理和极其有用的技术，在实际例子中它们很容易被细节所淹没。这里给出的大部分有用的类，如链接表、数组、字符串、矩阵、图形类、关联数组等，在广泛可用的各种商品的和非商品资源中，都有可用的“防弹”和/或“金盘”版本。那些“具有工业强度”的类和库中的许多东西，实际上不过是在这里可以找到的玩具版本的直接或间接后裔。

与本书的第1版相比，这一版更加强调本书的教学方面的作用。然而，这里的叙述仍然是针对有经验的程序员，并努力不去轻视他们的智慧和经验。有关设计问题的讨论有了很大的扩充，作为对读者在语言特征及其直接应用之外的要求的一种回应。技术细节和精确性也有所增强。特别是，这里的参考手册表现了在这个方向上多年的工作。我的目标是提供一本具有足够深度的书籍，使大部分程序员能在多次阅读中都有所收获。换句话说，这本书给出的是C++ 语言，它的基本原理，以及使用时所需要的关键性技术。欢迎欣赏！

致谢

除了在第1版序中致谢里所提到人们之外，我还要感谢Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot和Peter Weinberger, 他们对第2版的初稿提出了许多意见。许多人对C++从1985年到1991年的开发有很大影响，我只能提出其中几个：Andrew Koenig, Brian Kernighan, Doug McIlroy和Jonathan Shopiro。还要感谢参考手册“外部评阅”的许多参与者，以及在X3J16的整个第一年里一直在其中受苦的人们。

Bjarne Stroustrup
Murray Hill, 新泽西

第1版序

语言磨砺了我们思维的方式，
也决定着我們思考的范围。

——B.L. Whorf

C++ 是一种通用的程序设计语言，其设计就是为了使认真的程序员工作得更愉快。除了一些小细节之外，C++ 是C程序设计语言的一个超集。C++ 提供了C所提供的各种功能，还为定义新类型提供了灵活而有效的功能。程序员可以通过定义新类型，使这些类型与应用中的概念紧密对应，从而把一个应用划分成许多容易管理的片段。这种程序构造技术通常被称为数据抽象。某些用户定义类型的对象包含着类型信息，这种对象就可以方便而安全地用在那种对象类型无法在编译时确定的环境中。使用这种类型的对象的程序通常被称为是基于对象的。如果用得好，这些技术可以产生出更短、更容易理解，而且也更容易管理的程序。

C++ 里的最关键概念是类。一个类就是一个用户定义类型。类提供了对数据的隐藏，数据的初始化保证，用户定义类型的隐式类型转换，动态类型识别，用户控制的存储管理，以及重载运算符的机制等。在类型检查和表述模块性方面，C++提供了比C好得多的功能。它还包含了许多并不直接与类相关的改进，包括符号常量、函数的在线[⊖]替换、默认函数参数、重载函数名、自由存储管理运算符，以及引用类型等。C++ 保持了C高效处理硬件基本对象（位、字节、字、地址等）的能力。这就使用户定义类型能够在相当高的效率水平上实现。

C++ 及其标准库也是为了可移植性而设计的。当前的实现能够在大多数支持C的系统上运行。C的库也能用于C++ 程序，而且大部分支持C程序设计的工具也同样能用于C++。

本书的基本目标就是帮助认真的程序员学习这个语言，并将它用于那些非平凡的项目。书中提供了有关C++ 的完整描述，许多完整的例子，以及更多的程序片段。

致谢

如果没有许多朋友和同事持之以恒的使用、建议和建设性的批评，C++ 绝不会像它现在这样成熟。特别地，Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Rechie, Larry Rosler, Jerry Schwarz和Jon Shopiro对语言发展提供了重要的思想。Dave Presotto写出了流I/O库的当前实现。

此外，还有成百的人们对C++ 及其编译器的开发做出了贡献：他们给我提出改进的建议，描述他们所遇到的问题，告诉我编译中的错误等。我只能提出其中的很少几位：Gary Bishop, Abdreu Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult和Gary Walker。

⊖ inline在本书中统一地直译为“在线”，表示函数代码在调用位置的在线展开（而不是编译为独立的函数代码段）和运行中的在线执行（不经过函数调用）。目前常见的另一译法是“内联”，因其不妥，本书没有采用。
——译者注

许多人在本书的撰写过程中为我提供了帮助，特别值得提出的是Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, 以及参加1985年7月26 ~ 27日俄亥俄州哥伦布贝尔实验室C++ 课程的人们。

Bjarne Stroustrup
Murray Hill, 新泽西

目 录

出版者的话
专家指导委员会
中文版序
译者序
序
第2版序
第1版序

导 论

第1章 致读者	3
1.1 本书的结构	3
1.1.1 例子和参考	4
1.1.2 练习	5
1.1.3 有关实现的注记	5
1.2 学习C++	6
1.3 C++ 的设计	7
1.3.1 效率和结构	8
1.3.2 哲学注记	9
1.4 历史注记	9
1.5 C++ 的使用	11
1.6 C和C++	12
1.6.1 给C程序员的建议	13
1.6.2 给C++程序员的建议	13
1.7 有关在C++里编程的思考	14
1.8 忠告	15
1.9 参考文献	16
第2章 C++概览	19
2.1 为什么是C++	19
2.2 程序设计范型	19
2.3 过程式程序设计	20
2.3.1 变量和算术	21
2.3.2 检测和循环	22
2.3.3 指针和数组	23
2.4 模块程序设计	23
2.4.1 分别编译	24

2.4.2 异常处理	25
2.5 数据抽象	26
2.5.1 定义类型的模块	27
2.5.2 用户定义类型	28
2.5.3 具体类型	29
2.5.4 抽象类型	31
2.5.5 虚函数	33
2.6 面向对象的程序设计	33
2.6.1 具体类型的问题	33
2.6.2 类层次结构	34
2.7 通用型程序设计	36
2.7.1 容器	36
2.7.2 通用型算法	37
2.8 附言	38
2.9 忠告	39
第3章 标准库概览	40
3.1 引言	40
3.2 Hello, world!	40
3.3 标准库名字空间	41
3.4 输出	41
3.5 字符串	42
3.5.1 C风格的字符串	44
3.6 输入	44
3.7 容器	46
3.7.1 向量—— <i>vector</i>	46
3.7.2 范围检查	47
3.7.3 表—— <i>list</i>	48
3.7.4 映射—— <i>map</i>	49
3.7.5 标准容器	49
3.8 算法	50
3.8.1 迭代器的使用	51
3.8.2 迭代器类型	52
3.8.3 迭代器和I/O	53
3.8.4 遍历和谓词	54

3.8.5 使用成员函数的算法	56	5.3 到数组的指针	83
3.8.6 标准库算法	56	5.3.1 在数组里漫游	83
3.9 数学	57	5.4 常量	85
3.9.1 复数	57	5.4.1 指针和常量	87
3.9.2 向量算术	57	5.5 引用	88
3.9.3 基本数值支持	58	5.6 指向 <code>void</code> 的指针	90
3.10 标准库功能	58	5.7 结构	91
3.11 忠告	58	5.7.1 类型等价	94
第一部分 基本功能		5.8 忠告	94
第4章 类型和声明	63	5.9 练习	94
4.1 类型	63	第6章 表达式和语句	96
4.1.1 基本类型	64	6.1 一个桌面计算器	96
4.2 布尔量	64	6.1.1 分析器	96
4.3 字符类型	65	6.1.2 输入函数	100
4.3.1 字符文字量	66	6.1.3 低级输入	102
4.4 整数类型	66	6.1.4 错误处理	103
4.4.1 整数文字量	66	6.1.5 驱动程序	104
4.5 浮点类型	67	6.1.6 头文件	104
4.5.1 浮点文字量	67	6.1.7 命令行参数	105
4.6 大小	68	6.1.8 有关风格的注记	106
4.7 <code>void</code>	69	6.2 运算符概览	107
4.8 枚举	69	6.2.1 结果	109
4.9 声明	71	6.2.2 求值顺序	110
4.9.1 声明的结构	72	6.2.3 运算符优先级	110
4.9.2 声明多个名字	73	6.2.4 按位逻辑运算符	111
4.9.3 名字	73	6.2.5 增量和减量	112
4.9.4 作用域	74	6.2.6 自由存储	113
4.9.5 初始化	75	6.2.7 显式类型转换	116
4.9.6 对象和左值	76	6.2.8 构造函数	117
4.9.7 <code>typedef</code>	76	6.3 语句概览	118
4.10 忠告	77	6.3.1 声明作为语句	119
4.11 练习	77	6.3.2 选择语句	119
第5章 指针、数组和结构	79	6.3.3 迭代语句	122
5.1 指针	79	6.3.4 <code>goto</code>	123
5.1.1 零	80	6.4 注释和缩进编排	123
5.2 数组	80	6.5 忠告	125
5.2.1 数组初始化	80	6.6 练习	125
5.2.2 字符串文字量	81	第7章 函数	128
		7.1 函数声明	128

7.1.1 函数定义	128
7.1.2 静态变量	129
7.2 参数传递	130
7.2.1 数组参数	131
7.3 返回值	132
7.4 重载函数名	133
7.4.1 重载和返回类型	135
7.4.2 重载与作用域	135
7.4.3 手工的歧义性解析	135
7.4.4 多参数的解析	136
7.5 默认参数	137
7.6 未确定数目的参数	138
7.7 指向函数的指针	139
7.8 宏	143
7.8.1 条件编译	145
7.9 忠告	145
7.10 练习	146
第8章 名字空间和异常	148
8.1 模块化和界面	148
8.2 名字空间	150
8.2.1 带限定词的名字	151
8.2.2 使用声明	152
8.2.3 使用指令	153
8.2.4 多重界面	154
8.2.5 避免名字冲突	157
8.2.6 名字查找	159
8.2.7 名字空间别名	159
8.2.8 名字空间组合	160
8.2.9 名字空间和老代码	163
8.3 异常	166
8.3.1 抛出和捕捉	167
8.3.2 异常的辨识	168
8.3.3 在计算器中的异常	169
8.4 忠告	173
8.5 练习	173
第9章 源文件和程序	175
9.1 分别编译	175
9.2 连接	176
9.2.1 头文件	178

9.2.2 标准库头文件	179
9.2.3 单一定义规则	180
9.2.4 与非C++代码的连接	182
9.2.5 连接与指向函数的指针	184
9.3 使用头文件	184
9.3.1 单一头文件	184
9.3.2 多个头文件	187
9.3.3 包含保护符	191
9.4 程序	192
9.4.1 非局部变量的初始化	192
9.5 忠告	194
9.6 练习	194

第二部分 抽象机制

第10章 类	199
10.1 引言	199
10.2 类	199
10.2.1 成员函数	200
10.2.2 访问控制	201
10.2.3 构造函数	202
10.2.4 静态成员	203
10.2.5 类对象的复制	204
10.2.6 常量成员函数	205
10.2.7 自引用	205
10.2.8 结构和类	208
10.2.9 在类内部的函数定义	210
10.3 高效的自定义类型	210
10.3.1 成员函数	212
10.3.2 协助函数	214
10.3.3 重载的运算符	215
10.3.4 具体类型的意义	215
10.4 对象	216
10.4.1 析构函数	216
10.4.2 默认构造函数	217
10.4.3 构造和析构	218
10.4.4 局部变量	218
10.4.5 自由存储	220
10.4.6 类对象作为成员	221
10.4.7 数组	223

10.4.8 局部静态存储	224	12.1 引言	268
10.4.9 非局部存储	225	12.2 派生类	269
10.4.10 临时对象	226	12.2.1 成员函数	271
10.4.11 对象的放置	228	12.2.2 构造函数和析构函数	272
10.4.12 联合	229	12.2.3 复制	273
10.5 忠告	230	12.2.4 类层次结构	273
10.6 练习	230	12.2.5 类型域	274
第11章 运算符重载	233	12.2.6 虚函数	276
11.1 引言	233	12.3 抽象类	278
11.2 运算符函数	234	12.4 类层次结构的设计	280
11.2.1 二元和一元运算符	235	12.4.1 一个传统的层次结构	280
11.2.2 运算符的预定义意义	236	12.4.2 抽象类	283
11.2.3 运算符和用户定义类型	236	12.4.3 其他实现方式	285
11.2.4 名字空间里的运算符	237	12.4.4 对象创建的局部化	287
11.3 一个复数类型	238	12.5 类层次结构和抽象类	289
11.3.1 成员运算符和非成员运算符	238	12.6 忠告	289
11.3.2 混合模式算术	239	12.7 练习	289
11.3.3 初始化	240	第13章 模板	292
11.3.4 复制	241	13.1 引言	292
11.3.5 构造函数和转换	242	13.2 一个简单的String模板	293
11.3.6 文字量	243	13.2.1 定义一个模板	294
11.3.7 另一些成员函数	243	13.2.2 模板实例化	295
11.3.8 协助函数	244	13.2.3 模板参数	296
11.4 转换运算符	245	13.2.4 类型等价	296
11.4.1 歧义性	246	13.2.5 类型检查	297
11.5 友元	248	13.3 函数模板	298
11.5.1 友元的寻找	249	13.3.1 函数模板的参数	299
11.5.2 友元和成员	250	13.3.2 函数模板的重载	300
11.6 大型对象	251	13.4 用模板参数描述策略	302
11.7 基本运算符	253	13.4.1 默认模板参数	303
11.7.1 显式构造函数	253	13.5 专门化	304
11.8 下标	255	13.5.1 专门化的顺序	306
11.9 函数调用	256	13.5.2 模板函数的专门化	307
11.10 间接	257	13.6 派生和模板	308
11.11 增量和减量	259	13.6.1 参数化和继承	309
11.12 一个字符串类	260	13.6.2 成员模板	310
11.13 忠告	265	13.6.3 继承关系	311
11.14 练习	265	13.7 源代码组织	312
第12章 派生类	268	13.8 忠告	314

13.9 练习	314
第14章 异常处理	316
14.1 错误处理	316
14.1.1 关于异常的其他观点	318
14.2 异常的组组	318
14.2.1 派生的异常	319
14.2.2 多个异常的组组	321
14.3 捕捉异常	321
14.3.1 重新抛出	322
14.3.2 捕捉所有异常	322
14.4 资源管理	324
14.4.1 构造函数和析构函数的使用	325
14.4.2 <i>auto_ptr</i>	326
14.4.3 告诫	328
14.4.4 异常和 <i>new</i>	328
14.4.5 资源耗尽	329
14.4.6 构造函数里的异常	331
14.4.7 析构函数里的异常	332
14.5 不是错误的异常	333
14.6 异常的描述	334
14.6.1 对异常描述的检查	335
14.6.2 未预期的异常	336
14.6.3 异常的映射	336
14.7 未捕捉的异常	338
14.8 异常和效率	339
14.9 处理错误的其他方式	340
14.10 标准异常	342
14.11 忠告	344
14.12 练习	344
第15章 类层次结构	346
15.1 引言和概述	346
15.2 多重继承	346
15.2.1 歧义性解析	348
15.2.2 继承和使用声明	349
15.2.3 重复的基类	350
15.2.4 虚基类	352
15.2.5 使用多重继承	354
15.3 访问控制	357
15.3.1 保护成员	359

15.3.2 对基类的访问	360
15.4 运行时类型信息	361
15.4.1 <i>dynamic_cast</i>	363
15.4.2 在类层次结构中漫游	365
15.4.3 类对象的构造与析构	367
15.4.4 <i>typeid</i> 和扩展的类型信息	368
15.4.5 RTTI的使用和误用	370
15.5 指向成员的指针	371
15.5.1 基类和派生类	373
15.6 自由存储	374
15.6.1 数组分配	375
15.6.2 虚构造函数	376
15.7 忠告	377
15.8 练习	377

第三部分 标准库

第16章 库组织和容器	381
16.1 标准库的设计	381
16.1.1 设计约束	382
16.1.2 标准库组织	383
16.1.3 语言支持	385
16.2 容器设计	386
16.2.1 专门化的容器和迭代器	386
16.2.2 有基类的容器	388
16.2.3 STL容器	391
16.3 向量	392
16.3.1 类型	393
16.3.2 迭代器	394
16.3.3 元素访问	395
16.3.4 构造函数	396
16.3.5 堆栈操作	399
16.3.6 表操作	401
16.3.7 元素定位	403
16.3.8 大小和容量	404
16.3.9 其他成员函数	406
16.3.10 协助函数	406
16.3.11 <i>vector<bool></i>	407
16.4 忠告	407
16.5 练习	408

第17章 标准容器	409	18.4.4 约束器、适配器和否定器	458
17.1 标准容器	409	18.5 非修改性序列算法	463
17.1.1 操作综述	409	18.5.1 对每个做—— <i>for_each</i>	463
17.1.2 容器综述	412	18.5.2 查找族函数	464
17.1.3 表示	413	18.5.3 计数	465
17.1.4 对元素的要求	413	18.5.4 相等和不匹配	466
17.2 序列	416	18.5.5 搜索	467
17.2.1 向量—— <i>vector</i>	416	18.6 修改性序列算法	467
17.2.2 表—— <i>list</i>	416	18.6.1 复制	468
17.2.3 双端队列—— <i>deque</i>	420	18.6.2 变换	469
17.3 序列适配器	421	18.6.3 惟一化	471
17.3.1 堆栈—— <i>stack</i>	421	18.6.4 取代	473
17.3.2 队列—— <i>queue</i>	422	18.6.5 删除	474
17.3.3 优先队列—— <i>priority_queue</i>	423	18.6.6 填充和生成	474
17.4 关联容器	425	18.6.7 反转和旋转	475
17.4.1 映射—— <i>map</i>	425	18.6.8 交换	476
17.4.2 多重映射—— <i>multimap</i>	433	18.7 排序的序列	476
17.4.3 集合—— <i>set</i>	434	18.7.1 排序	476
17.4.4 多重集合—— <i>multiset</i>	435	18.7.2 二分检索	477
17.5 拟容器	435	18.7.3 归并	478
17.5.1 串—— <i>string</i>	435	18.7.4 划分	479
17.5.2 值向量—— <i>valarray</i>	435	18.7.5 序列上的集合运算	479
17.5.3 位集合—— <i>bitset</i>	435	18.8 堆	480
17.5.4 内部数组	439	18.9 最小和最大	481
17.6 定义新容器	439	18.10 排列	482
17.6.1 散列映射—— <i>hash_map</i>	440	18.11 C风格算法	482
17.6.2 表示和构造	441	18.12 忠告	483
17.6.3 其他散列关联容器	446	18.13 练习	483
17.7 忠告	446	第19章 迭代器和分配器	485
17.8 练习	446	19.1 引言	485
第18章 算法和函数对象	449	19.2 迭代器和序列	485
18.1 引言	449	19.2.1 迭代器的操作	486
18.2 标准库算法综述	449	19.2.2 迭代器特征类—— <i>iterator_traits</i>	487
18.3 序列和容器	453	19.2.3 迭代器类别	488
18.3.1 输入序列	453	19.2.4 插入器	490
18.4 函数对象	454	19.2.5 反向迭代器	491
18.4.1 函数对象的基类	456	19.2.6 流迭代器	492
18.4.2 谓词	456	19.3 带检查迭代器	495
18.4.3 算术函数对象	458	19.3.1 异常、容器和算法	499

19.4 分配器	500	21.2.3 用户定义类型的输出	538
19.4.1 标准分配器	500	21.3 输入	540
19.4.2 一个用户定义分配器	503	21.3.1 输入流	540
19.4.3 广义的分配器	505	21.3.2 内部类型的输入	540
19.4.4 未初始化的存储	506	21.3.3 流状态	542
19.4.5 动态存储	508	21.3.4 字符的输入	544
19.4.6 C风格的分配	509	21.3.5 用户定义类型的输入	546
19.5 忠告	510	21.3.6 异常	547
19.6 练习	510	21.3.7 流的联结	548
第20章 串	511	21.3.8 哨位	549
20.1 引言	511	21.4 格式化	550
20.2 字符	511	21.4.1 格式状态	550
20.2.1 字符特征类—— <i>char_traits</i>	512	21.4.2 整数输出	552
20.3 基础串类—— <i>basic_string</i>	513	21.4.3 浮点数输出	552
20.3.1 类型	514	21.4.4 输出域	553
20.3.2 迭代器	515	21.4.5 域的调整	555
20.3.3 元素访问	516	21.4.6 操控符	555
20.3.4 构造函数	516	21.5 文件流与字符串流	560
20.3.5 错误	517	21.5.1 文件流	561
20.3.6 赋值	518	21.5.2 流的关闭	562
20.3.7 到C风格字符串的转换	519	21.5.3 字符串流	563
20.3.8 比较	521	21.6 缓冲	564
20.3.9 插入	522	21.6.1 输出流和缓冲区	565
20.3.10 拼接	523	21.6.2 输入流和缓冲区	566
20.3.11 查找	524	21.6.3 流和缓冲区	567
20.3.12 替换	525	21.6.4 流缓冲区	567
20.3.13 子串	526	21.7 现场	571
20.3.14 大小和容量	527	21.7.1 流回调	572
20.3.15 I/O操作	527	21.8 C输入/输出	573
20.3.16 交换	528	21.9 忠告	575
20.4 C标准库	528	21.10 练习	576
20.4.1 C风格字符串	528	第22章 数值	578
20.4.2 字符分类	530	22.1 引言	578
20.5 忠告	530	22.2 数值的限制	578
20.6 练习	531	22.2.1 表示限制的宏	580
第21章 流	533	22.3 标准数学函数	580
21.1 引言	533	22.4 向量算术	582
21.2 输出	534	22.4.1 <i>valarray</i> 的构造	582
21.2.1 输出流	535	22.4.2 <i>valarray</i> 的下标和赋值	583
21.2.2 内部类型的输出	536	22.4.3 成员操作	584

22.4.4 非成员函数	586	24.2.2 忽视继承	638
22.4.5 切割	587	24.2.3 忽视静态类型检查	638
22.4.6 切割数组—— <i>slice_array</i>	589	24.2.4 忽视程序设计	641
22.4.7 临时量、复制和循环	593	24.2.5 排他性地使用类层次结构	642
22.4.8 广义切割	595	24.3 类	643
22.4.9 屏蔽	596	24.3.1 类表示什么	643
22.4.10 间接数组—— <i>indirect_array</i>	596	24.3.2 类层次结构	644
22.5 复数算术	597	24.3.3 包容关系	648
22.6 通用数值算法	599	24.3.4 包容和继承	649
22.6.1 累积—— <i>accumulate</i>	599	24.3.5 使用关系	653
22.6.2 内积—— <i>inner_product</i>	600	24.3.6 编入程序里的关系	654
22.6.3 增量变化	600	24.3.7 类内的关系	656
22.7 随机数	602	24.4 组件	661
22.8 忠告	603	24.4.1 模板	663
22.9 练习	603	24.4.2 界面和实现	665
		24.4.3 肥大的界面	667
		24.5 忠告	668
第四部分 用C++ 做设计		第25章 类的作用	670
第23章 开发和设计	607	25.1 类的种类	670
23.1 概述	607	25.2 具体类型	672
23.2 引言	607	25.2.1 具体类型的重用	672
23.3 目的与手段	609	25.3 抽象类型	674
23.4 开发过程	611	25.4 结点	676
23.4.1 开发循环	613	25.4.1 修改界面	677
23.4.2 设计目标	615	25.5 动作	680
23.4.3 设计步骤	616	25.6 界面类	681
23.4.4 试验和分析	623	25.6.1 调整界面	683
23.4.5 测试	625	25.7 句柄类	684
23.4.6 软件维护	625	25.7.1 句柄上的操作	687
23.4.7 效率	626	25.8 应用框架	688
23.5 管理	626	25.9 忠告	689
23.5.1 重用	627	25.10 练习	690
23.5.2 规模	628		
23.5.3 个人	629	附录和索引	
23.5.4 混成设计	630	附录A 语法	695
23.6 带标注的参考文献	631	附录B 兼容性	713
23.7 忠告	633	附录C 技术细节	724
第24章 设计和编程	635	附录D 现场	759
24.1 概述	635	附录E 标准库的异常时安全性	815
24.2 设计和程序设计语言	635	索引	845
24.2.1 忽视类	637		

导 论

这个导论是对C++ 语言及其标准库的主要概念和特征的综述，也是对本书的综述，还解释了这里所采用的描述语言概念及其使用的方式。除此之外，导论的各章还提供了有关C++、C++的设计以及C++使用情况的某些背景性信息。

章目

- 第1章 致读者
- 第2章 C++ 概览
- 第3章 标准库概览

“……而你，马库斯，你已经给了我许多东西；现在我要给你这个极好的忠告。做一个普通的人。放弃总是扮演马库斯·克可查的那个游戏。你为马库斯·克可查操心操得太多了，以致你已经变成了他的奴隶和囚徒。你在做任何事情之前都要首先考虑它将如何影响马库斯·克可查的幸福和声望。你总是非常害怕马库斯可能会做一点愚蠢的事，或者令人讨厌的事。这些真的那么有意义吗？整个世界的人都在做蠢事……我真心希望你能轻松一点，让你那小小的心回到轻松的状态。你必须从现在开始，去做不止一个人，做许许多多的人们，就像你可能想到的那么多……”

——卡伦·布利克森

（“梦想者”，出自“七个哥特人的传说”，

笔名伊萨克·迪尼森，

Random House Inc.

版权所有，Isak Dinesen, 1934年, 1961年修订）

第1章 致 读 者

“是时候了，”海象说，“该谈谈各种各样的事情了。”

——L. Carroll

本书的结构——怎样学习C++——C++的设计——效率和结构——哲学注记——历史注记——C++适合做什么——C和C++——给C程序员的建议——给C++程序员的建议——有关在C++里编程的思考——忠告——参考文献

1.1 本书的结构

这本书包括六个部分：

导论：第1~3章给出的是有关C++ 语言，它所支持的关键性程序设计风格，以及有关C++ 标准库的综述。

第一部分：第4~9章是有关C++ 内部类型，以及由它们出发构造程序的基本功能的一个具有教材形式的介绍。

第二部分：第10~15章是有关使用C++ 做面向对象和通用型程序设计的 一个具有教材形式的介绍。

第三部分：第16~22章介绍C++的标准库。

第四部分：第23~25章讨论设计和软件开发方面的一些论题。

附录：附录A~E提供了语言的技术性细节。

第1章是对全书的综述。这里提供了一些有关如何使用C++的建议，以及一些有关C++ 及其应用的背景性信息。你应该大略地读一读这一章，先注意读那些看起来有意思的内容，在读了本书的其他一些部分之后再回来读一读。

第2章和第3章是有关C++ 程序设计语言及其标准库的主要概念和特征的综述。这两章的目的是促使你在基础性概念和基本语言特征上用一些时间，在这里展示了利用完整的C++ 语言可以描述些什么。即使没有其他内容，这两章也会使你确信C++ 并不（只）是C，而且，在本书的第1版和第2版之后，C++ 已经走过了很长的一段路。第2章给出了有关C++ 的一个高层次的认识，其中的讨论集中在那些支持数据抽象、而向对象的程序设计和通用型程序设计的语言特征方面。第3章介绍了标准库的基本原理和主要功能，这也使我可以在随后的章节里使用标准库，也使你能在练习中利用各种库功能，而不是去直接依靠低级的内部特征。

作为导论的这些章也是一种具有一般性的技术的实例，这种技术将在这整本书中始终如一地运用：在展开对某些技术或者特征的更直接更实质性的讨论时，我有时会先简洁地说明一个概念，而到晚些时候再去深入地讨论它。这种方式使我可以在更一般性地处理一个题目之前给出很实际的例子。也就是说，本书的组织方式反映了这样的一种观点：我们通常的学

习最好是从具体发展到抽象——甚至是在那些抽象的东西本身很简单，回过头看非常明显的地方。

第一部分描述的是C++的一个子集，它支持传统上在C或Pascal里进行的那种风格的程序设计。这里的内容覆盖了基本类型、表达式、以及C++程序的控制结构。也讨论了由名字空间、源程序文件和异常处理所支持的模块化问题。本书假定你已熟悉在第一部分中用到的那些基本程序设计概念。例如，我将解释C++中表述循环和递归的功能，但却不去花许多时间解释为什么这些概念非常有用。

第二部分描述C++里定义和使用新类型的功能。具体的和抽象的类（界面）都在这里讨论（第10、12章），还有运算符重载（第11章），多态性，以及类层次结构的使用（第12、15章）。第13章讨论模板，也就是在C++里定义一族类型或函数的机制。这里阐释了提供容器（例如表），以及支持通用型程序设计的那些基本技术。第14章描述异常处理，讨论对错误的处理，说明有关容错的策略等。假定你或者是不很熟悉面向对象程序设计和通用型程序设计，或者是能从有关C++怎样支持主要的数据抽象技术的解释中获益。正因为此，我将不仅描述支持这些抽象技术的语言特征，也要解释这些技术本身。第四部分将在这个方向上继续前进。

第三部分描述C++标准库。这里的目标是为如何使用这个库提供一种理解，阐述一般性的设计和编程技术，也说明如何去扩充这个库。标准库提供了容器（如list、vector和map；第16、17章），标准算法（如sort、find和merge；第18、19章），字符串（第20章），输入输出（第21章），以及对数值计算的支持（第22章）。

第四部分讨论的是在把C++用于大型软件系统的设计和实现时所引出的论题。第23章集中于设计和管理方面，第24章讨论C++程序设计语言和设计问题之间的关系，第25章给出将类应用于设计的一些方法。

附录A是C++的语法描述，带有少量的标注。附录B讨论C和C++，以及标准C++（也称为ISO C++或ANSI C++）和在此之前的C++版本之间的关系。附录C描述一些语言技术实例。附录D解释标准库中支持国际化的功能。附录E讨论标准库在异常时安全性方面的保证和要求。

1.1.1 例子和参考

本书强调的是程序的组织，而不是算法的书写。因此，我完全避免了巧妙的或难以理解的例子。一个平凡的程序往往更适于用来展示语言定义的某一个方面，或者程序结构中的某一点。例如，我可能在某个地方采用Shell排序，而在实际代码中用快速排序则更好一些。我也经常把用更合适的算法重新实现作为一道练习题。在实际代码中，调用一个库函数也常比使用在这里展示语言特征的代码更值得称道。

教科书上的例子必然会给人有关软件开发的一种经过包装的观点。由于小例子的清晰性和简单性，由规模而引起的复杂性消失了。依我看，没有什么东西能够代替去写一些实际大小的程序，只有那样才能真正感受程序设计和程序设计语言究竟是什么。这本书将集中关注语言特征，关注那些支持组合出各种程序的基本技术，以及有关组合的规则。

有关示例的选择反映了我在编译器、基础库和模拟方面的研究背景。这些例子都是在真实代码中能找到的内容的简化版本。这种简化是必需的，只有这样才能保证语言特征和设计观点不会被繁琐细节所淹没。不存在没有真实背景的“灵巧”示例。在任何地方，只要可能，

我就把那一类实例移交给附录C，在那里的例子中总用变量名字*x*和*y*，类型名总是*A*和*B*，函数总是*f()*和*g()*。

在所有代码示例中，标识符用的是比例宽度的字体。例如，

```
#include<iostream>

int main()
{
    std::cout << "Hello, new world!\n";
}
```

初看起来这种表示风格好像“不大自然”，因为程序员已经习惯于采用等宽字体的代码。但是，一般的认识是，比例宽度的字体在表现正文方面优于等宽字体。采用比例宽度字体也使我的代码里更少出现非逻辑的断行现象。进一步说，我的实际经验说明，大部分人经过一小段时间后就会觉得这种风格更容易读。

只要有可能，C++ 语言和库特征都在将使用它们的环境中介绍，而不是在手册中以干巴巴的方式介绍。这里所描述的语言特征及其细节程度，也反映了我对于有效使用C++ 的需要的认识。与此伴生的另一本书，《*The Annotated C++ Language Standard*》（《带标注的C++ 语言标准》）将由Andrew Koenig和我合著，那是这个语言的一个完整定义，并附加一些注释，使之更容易理解。逻辑上说还应该有另一本伴生的书，《*The Annotated C++ Standard Library*（带标注的C++标准库》）。但是，由于时间和写作能力的限制，我无法允诺去完成这件事情。

本书内的相互参考采用2.3.4节（表示第2章第3节中的第4小节）、B.5.6节（表示附录B第5节中的第6小节），以及6.6[10]（第6章练习10）的形式。楷体用于表示强调（如“一个字符串文字量是不能接受的”），重要概念的第一次出现（如多态性）。C++ 语法中的非终结符用斜体（如*for-statement*）。黑斜体用于引用代码实例中的标识符、关键字和数值（如*class*、*counter*和*1712*）。

1.1.2 练习

练习被安排在各章的最后。这些练习主要是写某个程序的变形。请读者一定写出有关一个解的充分的代码，经过编译并至少用一些测试情况去运行它。这些练习的难度有相当大的变化，因此为它们加了有关其难度估计的标记。度量方式是指数的，如果一个(*1)练习需要花掉你10分钟时间，那么一个(*2)就可能需要一个小时，而(*3)可能要用一天。写程序和测试程序所需要的时间将更多地依赖于你的个人经验，而不是练习本身。如果你第一次去熟悉一个计算机系统，并要设法让一个(*1)练习运行，它可能耗费你一天的时间。在另一方面，如果某人正好手头有一组合适的程序，完成一个(*5)练习或许只要一个小时就够了。

任何有关C程序设计的书籍都可以作为第一部分额外练习的来源。任何有关数据结构或者算法的书都可以作为第二部分和第三部分的练习来源。

1.1.3 有关实现的注记

本书中使用的语言是在C++标准[C++, 1998]中定义的“纯的C++”。因此，这里的例子应该能在每个C++实现上运行。书中主要的程序片段都在几个C++实现里测试过。那些用到了最近纳入C++特征的程序未必能在每个C++实现上编译。当然，我看不出有什么必要去说明

哪个实现无法编译哪些例子。这种信息将会迅速过时，因为实现者们正在努力工作，以保证他们的实现能够正确地接受每一个C++ 特征。参看附录B，那里有一些关于如何处理老的C++ 编译器和为C编译器写的代码的建议。

1.2 学习C++

在学习C++ 时，最重要的事情就是集中关注概念，不要迷失在语言的技术细节中。学习语言的目的是成为一个更好的程序员；也就是说，使自己在设计和实现新系统时，以及在维护老系统时，能够工作得更有成效。为此，对于程序设计和设计技术的理解远比对细节的理解更重要，而这种理解的根本是时间和实践。

C++ 支持多种不同的程序设计风格。所有这些的基础是强类型检查，大部分的目标都是要获得一种高层次的抽象，以直接表达程序员的思想。每种风格都可以在有效管理时间和空间的情况下达到它的目的。来自别的不同语言（比如说C、Fortran、Smalltalk、Lisp、ML、Ada、Eiffel、Pascal或者Modula-2）的程序员应该认识到，要想从C++ 中获益，他们就必须花时间去学习，以使适合于C++ 的程序设计风格和技术真正变成自己的东西。这一建议同样也适用于那些熟悉C++ 早期的和表达能力较弱的版本的程序员们。

盲目地将一种在某个语言中很有效的技术应用到另一个语言，经常会导致笨拙的、性能低下的和难以维护的代码。在写这样的代码时也会备受挫折，因为每行代码和每条编译错误信息都在提醒程序员，正在使用的这个语言是与“那个老语言”不同的。你可以用Fortran、C、Smalltalk等的风格写程序，可以在任何语言里这样做。但是在一个有着不同设计哲学的语言里，这样做既不愉快也不经济。每种语言都可以是如何写C++ 程序的一个丰富的思想源泉。但是，这些思想必须转化为某种能够适应C++ 的一般结构和类型系统的东西，以便能够有效地出现在不同的环境中。去颠覆一个语言的基本类型系统，可能得到的至多是伊皮鲁斯式的胜利^①。

C++ 支持一种逐步推进的学习方式。你学习一个新语言的方式依赖于你已经知道些什么，还依赖于你的学习目的。并没有一种适合于所有人的学习方式。假定你学习C++ 是为了成为一个更好的程序员和设计师。这就是说，假定你学习C++ 的意图并不简单地是为了再学一种新的语法形式，要去做某些已经习惯的事情，而是想学习一种构造系统的新的更好的方式。这件事只能逐渐完成，因为获得任何有意义的新技能都要花时间，需要经过实践。请想一想，要学会一种新的自然语言，或是学会演奏一种新的乐器要花多少时间。成为一个更好的系统设计师可能容易一些，也可能快一些，但绝不会容易到或者快到大部分人所希望的那种程度。

正因为这样，你将在还没有理解所有语言特征和技术之前就去使用C++——常常是去构造实际的系统。通过支持多种程序设计范型（第2章），C++ 支持在不同掌握程度上进行生产性的程序设计。每一种新的程序设计风格将为你的工具箱增加一种新工具，而且每种风格本身都是有效的，每种都能提高你作为程序员的效率。C++ 的组织方式使你能大致线性地学习它的概念，并在学习过程中不断得到实际收益。这是非常重要的，因为这就使你得到的利益大致与你付出的努力成比例。

① 伊皮鲁斯式的胜利，指付出了极大代价而取得的胜利。古希腊伊皮鲁斯国王比鲁斯（Pyrrhus）打败了罗马军队，但付出了极大的牺牲。因此有这种说法。——译者注

有关一个人是否应该在学习C++之前先学C的问题存在着长期争论。我坚定地认为最好的方式是直接学习C++。C++更安全，表达能力更强，而且减少了关心低级技术的需要。在你已经掌握了C和C++的公共子集和某些C++直接支持的高级技术之后，你会更容易去学习C中那些更诡秘的部分，而需要它们只是为了弥补C中高级功能的缺位。附录B是一个指南，可以帮助程序员从C++走向C，比如说，去处理那些作为遗产的代码。

存在着一些独立开发和发行的C++实现，有许多的工具、库和软件开发环境可以使用。成堆的教科书、手册、杂志、通信、电子公告板、邮件列表、会议和课程不断通知你有关C++的最新开发情况，有关它的使用、工具、库和实现等。如果你计划去认真地使用C++，我强烈地建议你去查阅这些资源。任何单独的东西都有其重点和倾向性，所以请参考至少两个来源。例如，参考[Barton, 1994]、[Booch, 1994]、[Henricson, 1997]、[Koenig, 1997]、[Martin, 1995]。

1.3 C++ 的设计

简单性是一个重要设计准则：如果在某个地方有一个选择，简化语言的定义或者简化编译器，那么我们一定选前者。当然，还有一个最重要的考虑是保持与C的高度兼容性[Koenig, 1989]、[Stroustrup, 1994] (附录B)，这也就排除了对C语法的清理。

C++没有内部的高级数据类型，也没有高级的基本操作。举例来说，C++没有提供带有求逆运算的矩阵类型，也没有带拼接运算的字符串类型。如果某个用户需要一个类型，那么可以在语言本身之中定义它。事实上，定义新的通用或者专用类型就是在C++里最基本的程序设计活动。一个设计良好的用户定义类型与一个内部类型之间的差异仅仅在于其定义的方式，而不在其使用方式。在第三部分描述的标准库提供了许多这样的类型及其使用的例子。从用户的观点看，在内部类型和由标准库提供的类型之间的差异非常小。

在C++的设计中，极力避免了那些即使不用也会带来运行时间或者空间额外开销的特征。例如，要求必须在每个对象里存储某种“簿记信息”的结构被拒绝了。所以，如果你定义了一种由两个16位的量组成的结构，它将能放进一个32位的寄存器里。

C++被设计为能使用传统的编译和运行时的环境，也就是那种在UNIX上的C程序设计环境。幸运的是，C++从来没有被束缚于UNIX，它只是简单地采用UNIX和C作为一种语言、库、编译器、连接器、执行环境等之间关系的模型。这种最小化的模型帮助C++在几乎每个计算平台上取得了成功。然而，在那些提供了更多有意义的支持的环境里，使用C++当然就更好了，像动态装载、增量编译、类型定义数据库等都能得到很好的收效，又不会影响语言本身。

C++的类型检查和数据隐藏特征依赖于编译时对程序的分析，以防止因为意外而破坏数据的情况。它们并不提供系统安全性^①或防止某些人有意地打破这些规则。它们当然可以随意使用而不会带来运行时额外的时间或空间开销。这种想法是很有用的，一种语言特征必须不仅是优美的，还是在真实程序的环境中能够负担起的东西。

有关C++设计的更系统和详尽的描述，请看[Stroustrup, 1994]。

① 这里的安全性是secrecy，与其他章节里讨论的安全性(safety，如附录E)是两个不同问题。safety讨论程序本身构造中的问题，而secrecy则常指程序与人的关系，如保护系统抵御人为破坏等。——译者注

1.3.1 效率和结构

C++ 是以C程序设计语言为基础开发出来的，除了少量例外，它继续维持了以C作为一个子集。作为基础语言，C++ 的C子集设计保证了在它的类型、运算、语句与计算机直接处理的对象（数、字符和地址）之间的紧密对应关系。除了`new`、`delete`、`dynamic_cast`，以及`throw`运算符和`try`-块(`try-block`)之外，C++ 的各种表达式和语句都不需要特殊的运行时支持。

C++ 可以使用与C一样的函数调用及返回序列——或者其他效率更高的方式。在这种相对有效的机制仍然被认为是代价过高之处，C++ 函数还可以采用在线替换，这样就使我们能在享受到函数记法上方便的同时，又不必付出任何运行时的开销。

C的一个初始目标是在大部分苛刻的系统程序设计中代替汇编语言。在设计C++ 时，也特别注意了不在这些已经取得的领域中做出任何妥协。在C和C++ 之间的差异，从根本上说，在于强调类型和结构的级别不同。C是有表达能力的和宽容的，而C++ 的表达能力更强。当然，为了获得这种增强的表达能力，你必须更多地关注对象的类型。知道了对象的类型，编译器就能正确处理表达式；如果不是这样，你就必须自己在令人难受的细节程度上去描述有关操作。知道了对象的类型，编译器就能检查错误；否则这些东西就会遗留下来直至测试阶段——或许留到更晚的时候。请注意，使用类型系统去检查函数的参数，去保护数据不被意外地破坏，去提供新的类型，去提供新的操作，如此等等，这些在C++ 里都没有增加任何运行中的时间或者空间开销。

C++ 中特别强调程序的结构，这反映了自C设计以来程序规模的增长情况。你可以通过玩命干做出一个小程序（比如说，1 000行），甚至是在你违反了所有有关好风格的规则的情况下。而对于更大的程序，情况就完全不同了。如果一个100 000行的程序的结构极其糟糕，你就会发现，引进新错误的速度像清除老错误一样快。C++ 的设计就是为了使较大的程序能够以一种合理的方式构造出来，并因此使一个人也有可能对付相当大的一批代码。进一步的目标是使一个平均行的C++ 代码能够表述出远比一个平均行的C或Pascal代码更多的东西。C++ 已经证明它超过了这些目标。

并不是每块代码都可能是结构良好的，与硬件无关的，容易阅读的，如此等等。C++ 也拥有一些特征，其意图就是为了以一种直截了当和高效的方式去操纵硬件功能，而不顾安全性或者容易理解诸方面的问题。它还拥有一些特征，使得我们能够将这样的代码隐藏在优美和安全的界面之后。

很自然，对更大型的程序使用C++ 语言，将会导致由成组的程序员使用C++。C++ 所强调的模块化，强类型的界面，以及灵活性在这里都能发挥作用。C++ 在各种为写大程序而提供的功能之间做了很好的平衡，像其他任何语言一样好。当然，随着程序变得更大，与它们的开发和维护相关的问题也会逐渐从语言的问题转移到更为全局性的工具和管理的问题。第四部分将探讨这方面的论题。

本书强调的是为提供通用功能、普遍有用的类型、库等的各种技术。这些技术能为写小程序的程序员服务，也能为写大程序的程序员服务。进一步说，由于任何非平凡的程序都是由许多半独立的部分组成，写这些部分的技术定能服务于开发所有应用的程序员。

你或许会怀疑，用更细节的类型结构去刻画大程序，会不会导致更大的程序正文？对于C++ 而言情况并不是这样。一个声明了函数参数类型、使用了类等的C++ 程序，通常比没有使用这些功能的等价C程序短一点。在那些使用了库的地方，C++ 程序就要比等价的C程序短得多。当然了，这里还得假定那个功能等价的C程序确实能构造出来。

1.3.2 哲学注记

一个程序设计语言要服务于两个相互关联的目的：它要为程序员提供一种描述所需执行的动作的载体；还要为程序员提供一组概念，使他们能利用这些概念去思考什么东西是能够做的。在理想的情况下，第一个用途要求一种“尽可能接近机器的”语言，以使机器的所有重要方面，都能以一种对程序员相当明显的方式简单而有效地加以处理。C语言的基本设计就是基于这一观点。而第二个用途所要求的理想语言是“尽可能接近需要解决的问题”，这样才能使解决方案中的概念能够直接而紧凑地表达出来。被加入C语言，从而塑造出C++的那些概念，从根本上说，就是基于这个观点设计的。

在我们思考/编程所用的语言和我们能够设想的问题与解之间的联系非常紧密。正是由于这个因素，以避免程序员犯错误为目的而对语言的特征加以限制，这一做法至少也是很危险的。就像自然语言中的情况，掌握至少两种语言就非常有价值。一个语言为程序员提供了一组概念工具；如果它们不适合于某件工作，程序员将简单地放弃这个语言。好的设计和不出现错误都不能仅由某些语言特征的存在或者不存在来保证。

类型系统对于各种各样非平凡的工作都特别有帮助。事实上，C++ 的类概念已经被证明是一种极为强有力的概念工具。

1.4 历史注记

我发明了C++，写出了它的第一个定义，做出了它的第一个实现。我选择并整理出C++的设计准则，设计了它的所有主要特征，并在C++ 标准化委员会里负责处理扩充建议。

很清楚，C++ 大大地受惠于C [Kernighan, 1978]。除封闭了其类型系统中的少量严重漏洞之外（附录B），C++仍保留C作为一个子集。我还保留了C在功能上的强项，能在足够低的层次上处理最苛刻的系统程序设计工作。C转而从其先驱BCPL [Richards, 1980] 受惠颇多；事实上，BCPL的 // 注释约定也被（重新）引进了C++。给C++ 以灵感的另一个主要来源是Simula67 [Dahl, 1970][Dahl, 1972]；类的概念（包括派生类和虚函数）都是在那里搬过来的。C++ 有关重载运算符和自由地将声明放置在可以出现语句的任何位置的功能，使人联想到Algol68 [Woodward, 1974]。

在本书的第一版之后，这个语言已经经过广泛的审查和精炼。审查的主要部分是对于重载的解析、连接，以及存储管理功能。此外还做了许多小修改，以增强与C的兼容性。还加进了一些推广和若干主要的扩充，包括：多重继承、*static*成员函数、*const*成员函数、*protected*成员、模板、异常处理、运行时类型识别和名字空间。所有这些扩充和修订的主旨都是为了使C++ 能够成为一个编写库、使用库的更好的语言。有关C++ 演化过程的描述参见[Stroustrup, 1994]。

模板功能的设计，从根本上说，是为了支持静态类型的容器（如表、向量和映射），以及优雅有效地使用这些容器（通用型程序设计）。这里的一个关键目标是减少宏和强制（显式类型转换）的使用。模板机制部分地受到Ada中类属^①的启发（包括其威力及其弱点），部分地

① 类属（generic）是Ada语言引进的，作为定义带类型参数的模块的机制。术语generic具有“类属的”、“一般的”之意，国内Ada文献中一直译为“类属”。本书中也大量出现generic这个词，指与模板机制有关的设计——一类具有通用性的程序体（函数、类）的机制，与“类”并无直接联系。为避免引起误解，也为了使其内在含义更加明显，本书中将始终把generic programming译为“通用型程序设计”。——译者注

受到Clu语言参数模块的影响。与此类似，C++ 的异常处理机制部分地受到Ada [Ichbiah, 1979]、Clu [Liskov, 1979] 和ML [Wikström, 1987] 的影响。其他方面的开发是在1985~1995年的时间跨度中做出的，例如，多重继承、纯虚函数，以及名字空间，这些基本上是在C++使用经验推动下推广而来，而不是由其他语言引进的。

这个语言的早期版本是大家都知道的“带类的C” [Stroustrup, 1994]，它从1980年开始使用。初始发明这个语言，是因为我想去写某些事件驱动的模拟程序，Simula67可能对于它们是最理想的，除了效率考虑之外。“带类的C”被用在一些主要的项目上，这使其用于写那种使用最少的时间和空间的程序的功能得到了严格的检验。这个语言缺乏运算符重载、引用、虚函数、模板、异常和许多细节。C++ 在研究组织之外的最初使用是在1983年。

名字C++（读作“see plus plus”^①）是Rick Mascitti在1983年夏天起的名字。这个名字象征着从C改变过来的演化性质；“++”是C的增量运算符。稍微短一点的名字“C+”则是个语法错误，它也曾被用于另一个与C++毫无关系的语言。C的语义权威们认为C++ 不如 ++C。它没有被称做D是因为它是C的一个扩充，而且也从未打算通过删除某些特征去修正一些问题。对于名字C++ 的另一种解释，请参看 [Orwell, 1949] 的附录。

C++ 原始的基本设计就是为了使我的朋友和我不必去用汇编、C或各种摩登的高级语言写程序。它的主要用途是为了那些作为个人的程序员，使他们能更容易和更愉快地写出好的程序来。在早期的那些年里根本就没有C++ 的纸面设计；设计、文档和实现是平行开展的。从来没有一个“C++项目”或者“C++设计委员会”。自始至终，C++的演化就是为了面对用户遇到的问题，也是在我的朋友、同事和我之间讨论的结果。

后来，C++ 使用的爆炸性增长导致情况产生了某些变化。在1987年的某个时候，情况已经很清楚，C++ 的标准化已是一件不可避免的事情了，我们需要开始去为标准化工作准备一个基础 [Stroustrup, 1994]。这导致了一种有意识的努力，去维持C++ 编译器的实现者们和主要用户之间的联系，通过文章和电子邮件，也通过在C++会议和其他各种场合中面对面的讨论。

AT&T的贝尔实验室对这个工作做出了主要的贡献，它允许我将C++ 参考手册的草稿和各种修订版本提供给实现者和用户共享。由于这些人中的许多是为那些可以看做是AT&T的竞争对手的公司工作的，这种分发的价值怎样估计都不过分。一个不那么光明磊落的公司可能早就导致了严重的语言分裂局面，为此它只要什么都不做也就够了。在那个时候，大约一百个人，来自数十个组织，阅读并评述着那本被广泛接受的参考手册，它也被作为ANSI C++ 标准化工作的基础文件。这些人的名字可以在《*The Annotated C++ Reference Manual*》（《带标注的C++ 参考手册》）[Ellis, 1989] 中找到。最后，ANSI的X3J16委员会于1989年12月在Hewlett-Packard的建议下建立起来。到1991年7月，这个ANSI（美国国内的）C++标准化变成了ISO（国际）的C++ 标准化工作的一部分。自1990年起，这个联合的C++标准化委员会就已经成为有关C++ 的演化和精练其定义的主要论坛。我自始至终在这些委员会中服务，特别是作为有关扩充的工作组的主席，我对有关C++ 的重要修改以及增加新语言特征的提议的处理直接负责。最初的标准草案在1995年4月提供给公众审阅，ISO C++标准（ISO/IEC 14882）在1998年被批准。

C++ 是与本书中描述的某些关键性的类携手一起演化前进的。例如，我设计了复数、向

① 中国人一般读作“C加加”。——译者注

量和堆栈类，以及运算符的重载机制；字符串和表类是Jonathan Shopen和我一起开发的，作为同一个工作的一部分。Jonathan的字符串和表类作为库的部分最早得到广泛使用。标准C++库的字符串类就植根于这些早期工作。在 [Stroustrup, 1987] 和12.7[11] 描述的作业库是曾经写出的最早“带类的C”程序。我写出了它及其一些相关的类，是为了支持具有Simula风格的模拟。这个作业库已经被修改并重新实现，主要是由Jonathan Shopen完成，并一直被广泛使用着。在本书的第一版里描述的流库是我设计和实现的；Jerry Schwarz将其转变为iostream库（第21章），采用了Andrew Koenig的操控符技术（21.4.6节）和其他思想。标准化过程中又对这个iostream库做了进一步精练，其中的大量工作是由Jerry Schwarz、Nathan Myers和Norihiro Kumagai完成的。模板功能的开发受到由我、Andrew Koenig、Alex Stepanov等人设计的vector、map、list和sort模板的影响。在另一方面，Alex Stepanov在使用模板的通用型程序设计方面的工作产生出标准C++ 库的容器和算法部分（16.3节、第17、18章、19.2节）。数值计算的valarray库第22章的基础是Kent Budge的工作。

1.5 C++ 的使用

C++ 被数以十万计的程序员应用到几乎每个领域中。这种应用得到十几个相互独立的实现，数以百计的库、数以百计的教科书、几种技术杂志、许多会议，以及不计其数的顾问们的支持。在各种层次上的培训和教育到处都可以获得。

早期的应用趋向于具有很强的系统程序设计色彩。例如，有几个主要操作系统是在C++里写出的 [Compbell, 1987] [Rozier, 1988] [Hamilton, 1993] [Berg, 1995] [Parrington, 1995]，更多系统用C++ 做了其中的关键部分。我认为C++不应在低层次的效率上妥协，这就使我们有可能会用C++ 写设备驱动程序，或者其他需要在实时约束下直接操作硬件的软件。在这样的代码中，性能的可预见性至少也与粗略的速度同样重要。C++的设计使得它的每种特征都可以在严格的时间和空间约束下使用 [Stroustrup, 1994, 4.5节]。

在大多数应用中都存在一些代码片段，它们在性能上的可接受性是至关重要的。当然，最大部分的代码并不在这些片段里。对于大部分代码而言，可管理、容易扩充、容易测试是最关键的问题。C++ 对所有这些关注点的支持已使它被广泛应用于一些领域，在其中的一些领域里可靠性是最基本的要求，在另一些领域里需求在不断地随着时间而发生显著的变化。这方面的例子如银行、贸易、保险业、远程通信，以及各种军事用途。许多年来，美国的长途电话系统的核心控制依赖于C++，所有800电话（即那些由被叫方付款的电话）都由一个C++ 程序进行路由 [Kamath, 1993]。许多这样的应用是大规模的，并且长期运行着。作为这种情况的结果，稳定性、兼容性和可伸缩性都成为C++ 开发中被始终关注的问题。成百万行的C++ 程序并不是罕见的情况。

与C类似，C++ 在设计时并没有将数值计算放在心里。但是，确实有许多数值的、科学的，以及工程的计算是在C++ 里做的。产生这种情况的一个主要原因是，传统的数值性工作常常必须与图形以及基于数据结构的计算相结合，而这些很难融进传统Fortran的模型之中 [Budge, 1992] [Barton, 1994]。图形学 and 用户界面正是使用C++ 最深入的领域。任何人要是使用过Apple Macintosh或者运行着Windows的PC，也都间接地使用了C++，因为这些系统的基本用户界面都是C++程序。此外，一些最流行的支持UNIX中X的库也是用C++ 写的。这样，C++就成了大量应用的最常见选择，只要用户界面是其中的主要部分。

所有这些都指向了或许是C++的最强之处：它能够有效地用到那些需要在各种各样的不同应用领域中工作的应用系统上。很容易找到一个应用系统，其中涉及到局域网络或者广域网络、数值处理、图形、与用户的交互，以及数据库访问等。在传统上，这些应用领域被认为是不同的，最常见的情况是它们由不同的技术团体使用各自的程序设计语言去处理。然而，C++ 已经被应用于所有的这些领域。进一步说，它还能与用其他语言写出的代码片段或者程序共存。

C++ 被广泛应用于教学和研究。这很令一些人吃惊，因为有些人曾经——正确地——指出，C++ 并不是已经设计出的各种语言中最小和最清晰的。但不管怎样，它是：

- 对于教授基本概念而言足够清晰的。
- 对于深刻的项目而言足够现实、高效和灵活的。
- 对依赖各种不同开发和执行环境的组织或研究机构而言，使用起来足够方便。
- 对作为教高级概念和技术的媒介而言，足够的容易理解。
- 对作为从学习到非学术使用的工具而言，也足够的商业化。

C++ 是一个可以伴随你成长的语言。

1.6 C和C++

C被选作C++ 的基础语言是因为它：

- [1] 是通用的、简洁的、相对低级的。
- [2] 适合用于大部分系统程序设计工作。
- [3] 可以在每个地方的任何系统上运行。
- [4] 适应于UNIX程序设计环境。

C有它的问题，但一个从空白出发设计的语言也必然会有一些问题，况且我们了解C的问题。更重要的是，从C出发已经使“带类的C”成了一个有用的工具（或许还比较笨拙），而且只是在开始想到要将类似Simula的类加到C上之后没几个月的时间。

随着C++ 的使用变得更为广泛，以及它所提供的覆盖和超越C的功能变得更加重要，关于是否还应该保持兼容性的问题又一再地被提出来。很清楚，如果抛弃C的某些传统就可以避免一些问题（参见，例如 [Sethi, 1981]）。这些都没有做，因为：

- [1] 存在着成百万行的C代码可能从C++ 中获益，先决条件是不必将它们完全从C重新写成C++。
- [2] 存在着成百万行用C写出的库函数和功能软件代码可以从C++ 程序里使用，或者在C++ 程序之上使用；先决条件是C++ 能够与C连接兼容，且在语法上与C类似。
- [3] 存在着数以十万计的程序员了解C语言，只需要去学习C++ 新特征的使用，而不想去重新学习基础。
- [4] C++ 和C将在许多年被同一一些人用于同样的系统，因此其差异必须或者是很小，或者是很大，以最大限度地减少错误和混乱的发生。

C++ 的定义已经做了许多修订，以保证任何同时在C和C++ 里合法的结构在两个语言中都具有同样的意义（除了少量例外；B.2节）。

C语言本身也在发展和演化，部分地是在C++ 开发的影响之下 [Rosler, 1984]。ANSI C标准 [C, 1990] 就包含了从“带类的C”借去的函数声明语法。借鉴是双向的，例如void*指针类

型是为ANSI C发明的，但却在C++ 里第一次实现。正如本书的第一版所允诺的，C++ 的定义已经过修订，以去掉无缘无故的不兼容性。今天的C++比原来更加与C兼容了。这里的想法是让C++ 尽可能接近ANSI C——但又不过于接近 [Koenig, 1989]。百分之百的兼容性从来就不是目标，因为这将危害类型安全性以及用户类型与内部类型的平滑集成。

了解C并不是学习C++ 的先决条件。在C中编程序被鼓励使用的许多技术和诀窍由于C++ 语言的特征而变得多余了。例如，显式类型转换（*casting*）在C++ 里就没有在C里那么频繁（1.6.1节）。然而，好的C程序倾向于也是C++程序。例如，在Kernighan和Ritchie的《*The C Programming Language*》，2nd Edition（《C程序设计语言》已由机械工业出版社翻译出版）[Kernighan, 1988] 里的每个程序都是C++ 程序。任何有关静态类型语言的经验对于学习C++ 也都能有所帮助。

1.6.1 给C程序员的建议

一个人对C了解得越好，在写C++程序时大概就越难避免C的风格，并会因此丢掉某些潜在C++的优势。请看一看附录B，那里描述了C与C++之间的差异。这里是几个有关的要点，在这些地方做同样的事情时，在C++ 里存在比C更好的方式：

- [1] 在C++里几乎不需要用宏。用***const***（5.4节）或***enum***（4.8节）定义明显的常量，用***inline***（7.1.1节）避免函数调用的额外开销，用***template***（第13章）去刻画一族函数或者类型，用***namespace***（8.2节）去避免名字冲突。
- [2] 不要在你需要变量之前去声明它，以保证你能立即对它进行初始化。声明可以出现在能出现语句的所有位置上（6.3.1节），可以出现在***for***语句（*for-statement*）的初始化部分（6.3.3节），也可以出现在条件中（6.3.2.1节）。
- [3] 不要用***malloc()***。***new***运算符（6.2.6节）能将同样的事情做得更好。对于***realloc()***，请试一试***vector()***（3.8节）。
- [4] 试着去避免***void****、指针算术、联合和强制，除了在某些函数或类实现的深层之外。在大部分情况下，强制都是设计错误的指示器。如果你必须使用某个显式的类型转换，请设法去用一个“新的强制”（6.2.7节），设法写出一个描述你想做的事情的更精确的语句。
- [5] 尽量少用数组和C风格的字符串。与传统的C风格相比，使用C++标准库***string***（3.5节）和***vector***（3.7.1节）常常可以简化程序设计。

如果要符合C的连接规则，一个C++函数就必须被声明为具有C连接的（9.2.4节）。

最重要的是，试着将程序考虑为一组由类和对象表示的相互作用的概念，而不是一堆数据结构和一些去拨弄数据结构中二进制位的函数。

1.6.2 给C++程序员的建议

到今天，许多人使用C++已经十几年了。大部分人是在某个单一的环境里使用C++，并已学会了在早期编译器和第一代的库所强加的束缚之下生存。经常可以看到这种情况，一个很有经验的C++程序员不仅很多年没有注意引进的新特征，也没有看到有关特征之间关系的变化，而这些情况已经使一些全新的程序设计技术变成可行的东西了。换句话说，你在第一次学习C++时没有想到或者认为不实际的东西，或许今天已经变成一种高明的方式。你只有通

过重新考察基础的东西才能弄清楚它们。

请按顺序浏览各章，如果你已经知道了某一章的内容，你可以只用几分钟就翻过去；如果你还不知道其内容，那么你会学到一些不曾预料到的东西。我在写这本书时就学到了不少东西，而且我怀疑会有哪个C++程序员知道这里给出的所有特征和技术。进一步说，要用好这个语言，你需要一种观点，以便给这组特征和技术带来一种秩序。通过书中的组织结构和实例，本书就提供了一种这样的观点。

1.7 有关在C++里编程的思考

在理想情况下，你需要通过三个步骤完成设计一个程序的工作。首先，你取得对问题的一个清晰的理解（分析）；而后你标识出在一个解决方案中所涉及的关键性概念（设计）；最后，你用一个程序表达这个解决方案（编程）。然而，问题的细节和解决方案中的概念，常常只有通过在一个程序中描述它们，以及让程序以可接受的方式运行的努力之下，才能真正被清楚地理解。而这正是程序设计语言选择的关键之处。

在大部分应用中都存在一些概念，它们很不容易表示为某个基本类型，也不容易表述为没有与之关联的数据的函数。遇到一个这样的概念，请在程序里声明一个类去表示它。一个C++类就是一个类型，也就是说，它刻画了这个类的对象的行为：它们如何建立，可以被如何操作，以及它们如何销毁。一个类可能也刻画了这些对象如何表示，虽然在设计一个程序的早期阶段这并不是一个主要考虑。写出好的程序，最关键的就是去设计这些类，使它们中的每一个都能很清楚地表示某个概念。这经常意味着你必须集中注意一些这样的问题：这个类的对象应该如何建立？这个类的对象能够被复制/销毁吗？什么操作能够作用于这种对象？如果对这类问题不存在很好的回答，对应的概念或许是从一开始就很不“清楚”。这时再多想想有关的问题以及为它所设定的解决方案，而不是立即开始去围绕着那个问题编码，这样做可能是个很好的主意。

最容易处理的概念是那些有着传统数学形式的东西：各种各样的数，集合，几何形状等。基于文本的I/O、字符串、基本容器、对这些容器的基本算法，以及一些数学类都是标准C++库的组成部分（第3章、16.1.2节）。除此之外，还存在着许多可以使用的支持通用的或者各种领域专用概念的令人眼花缭乱的库。

概念不会存在于真空之中；总是存在着相互关联的一簇簇的概念。将程序里各个类按它们之间的关系组织起来——即确定一个解决方案所涉及到的不同概念之间的准确关系——常常比首先单独地列出一些类更困难。最好别把结果弄成了一锅糨糊，其中的每一个类（概念）都相互依赖。考虑两个类，A和B。像“A调用B的函数”，“A建立起一些B”，“A包含一个B成员”之类的关系很少会造成重要的问题。而像“A使用B的数据”之类的关系通常应该能清除掉。

威力最大的一种管理复杂性的智力工具就是某种层次性的序关系，也就是说，将相互有关的概念组织到一个树形结构中，使最一般的概念成为树根。在C++里，派生类表示的就是这种结构。一个程序常常能组织为一组类，或者一组类的有向无环图。这时，程序员刻画一组基类，每个都有它的一组派生类。虚函数（2.5.5节、12.2.6节）常常能被用于为一个概念的最一般版本（一个基类）定义操作。如果有必要，可以针对特定的类（派生类），对这些操作的解释做进一步的精确化。

有时，甚至一个有向无环图看起来也不足以组织起一个程序里的概念；有些概念似乎具有内在的相互依赖性。在这种情况下，我们应设法将这种循环依赖关系局部化，使它们不会影响程序的整体结构。如果你无法清除这种相互依赖，也无法将其局部化，那么你很可能进入了一个困境，没有一种程序设计语言能够帮你跳出来。除非你能设想出在基本概念之间的某种很容易陈述的关系，否则那个程序多半会变得无法管理。

解开依赖图的一种最好的工具就是界面和实现的清晰分离。抽象类（2.5.4节、12.3节）是C++处理这种问题的基本工具。

共性的另一种形式可以通过模板（2.7节、第13章）表示。一个类模板刻画了一族类。例如，一个表模板刻画了“T的表”，其中T可以是任何类型。这样，模板就是这样一种机制，它刻画的是如何通过给定另一个类作为参数，就可以生成出一个新的类来。最常见的模板是容器类，例如表、数组和关联数组，以及使用这些容器的基本算法。将一个类及其相关函数的参数化通过一个使用继承的类型表达通常是个错误，这件事最好是用模板做。

应该记住，许多程序设计工作能够仅用基本类型、数据结构、普通函数和若干库类完成，这样做既简单又清晰。涉及到定义新类型的全套装备都不应该使用，除了在那些确实需要它们的地方以外。

问题“一个人怎样才能用C++里写出好的程序？”与问题“一个人怎样才能写出好的英语散文”类似。存在着两个回答：“了解你想说的是什么”以及“实践，模仿好的作品”。两者都适用于C++，就像它们适用于英语一样——去实践这一想法也同样不容易。

1.8 忠告

这里是一组在你学习C++的过程中或许应该考虑的“规则”。随着你变得更加熟练，你将能把它转化为某种更适合你的那类应用系统或者你自己的程序设计风格的东西。它们有意被写得很简单，因此都缺乏细节。请不要太拘泥于它们的字面意义。要写出一个好程序需要智慧、品味和耐性。你不会第一次就能把它搞好的。试验！

[1] 在编程序时，你是在为你针对某个问题的解决方案中的思想建立起一种具体表示。让程序的结构尽可能地直接反映这些思想：

- [a] 如果你能把“它”看成一个独立的概念，就把它做成一个类。
- [b] 如果你能把“它”看成一个独立的实体，就把它做成某个类的一个对象。
- [c] 如果两个类有共同的界面，将此界面做成一个抽象类。
- [d] 如果两个类的实现有某些显著的共同东西，将这些共性做成一个基类。
- [e] 如果一个类是一种对象的容器，将它做成一个模板。
- [f] 如果一个函数实现对某容器的一个算法，将它实现为对一族容器可用的模板函数。
- [g] 如果一组类、模板等互相之间有逻辑联系，将它们放进一个名字空间里。

[2] 在你定义一个并不是实现某个像矩阵或复数这样的数学对象的类时，或者定义一个低层的类型如链接表的时候：

- [a] 不要使用全局数据（使用成员）。
- [b] 不要使用全局函数。
- [c] 不要使用公用数据成员。
- [d] 不要使用友元，除非为了避免[a]或[c]。

[e] 不要在一个类里面放“类型域”[⊖]；采用虚函数。

[f] 不要使用在线函数，除非作为效果显著的优化。

更特殊或更详尽的实用规则可以在每章最后的“忠告”一节里找到。请记住，这些忠告只是粗略的实用规则，而不是万古不变的定律。它们只应使用在“合理的地方”。从来就没有任何东西能够替代智慧、经验、常识和好的鉴赏力。

我发现具有“绝不要做这个”形式的规则不大有帮助。因此，大部分忠告被写成应该做什么的建议，而否定性的建议也倾向于不采用绝对禁止的短语。据我所知，没有任何一种主要的C++特征没有被良好地使用过。在有关“忠告”的节里不包括解释，相反，每条忠告都引用了本书中某些适当的章节。在给出否定性的忠告时，对应章节里通常都提供了有关其他替代方式的建议。

1.9 参考文献

正文中有一些直接写出的参考文献，这里是一个不长的有关书籍和文章的列表，它们都直接或者间接地被提到过。

- [Barton,1994] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 1-201-53393-6.
- [Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38 No. 10. October 1995.
- [Booch,1994] Grady Booch: *Object-Oriented Analysis and Design*. Benjamin/Cummings. Menlo Park, Calif. 1994. ISBN 0-8053-5340-2.
- [Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.
- [C,1990] X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899. Computer and Business Equipment Manufacturers Association. Washington, DC, USA.
- [C++,1998] X3 Secretariat: *International Standard – The C++ Language*. X3J16-14882. Information Technology Council (NSITC). Washington, DC, USA.
- [Campbell,1987] Roy Campbell, et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.
- [Coplien,1995] James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. Reading, Mass. 1995. ISBN 1-201-60734-4.
- [Dahl,1970] O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language*. Norwegian Computing Center S-22. Oslo, Norway. 1970.
- [Dahl,1972] O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction in Structured Programming*. Academic Press, New York. 1972.
- [Ellis,1989] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.
- [Gamma,1995] Erich Gamma, et al.: *Design Patterns*. Addison-Wesley. Reading, Mass. 1995. ISBN 0-201-63361-2.
- [Goldberg,1983] A. Goldberg and D. Robson: *SMALLTALK-80 – The Language and Its Implementation*. Addison-Wesley. Reading, Mass. 1983.

⊖ 指那种为了说明一个类所存储数据的情况而放置的标志域。——译者注

- [Griswold,1970] R. E. Griswold, et al.: *The Snobol4 Programming Language*. Prentice-Hall. Englewood Cliffs, New Jersey. 1970.
- [Griswold,1983] R. E. Griswold and M. T. Griswold: *The ICON Programming Language*. Prentice-Hall. Englewood Cliffs, New Jersey. 1983.
- [Hamilton,1993] G. Hamilton and P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects*. Proc. 1993 Summer USENIX Conference. USENIX.
- [Henricson,1997] Mats Henricson and Erik Nyquist: *Industrial Strength C++: Rules and Recommendations*. Prentice-Hall. Englewood Cliffs, New Jersey. 1997. ISBN 0-13-120965-5.
- [Ichbiah,1979] Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14 No. 6. June 1979.
- [Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72 No. 5. September/October 1993.
- [Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*. Prentice-Hall. Englewood Cliffs, New Jersey. 1978.
- [Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language (Second Edition)*. Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.
- [Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1 No. 7. July 1989.
- [Koenig,1997] Andrew Koenig and Barbara Moo: *Ruminations on C++*. Addison Wesley Longman. Reading, Mass. 1997. ISBN 1-201-42339-1.
- [Knuth,1968] Donald Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Mass.
- [Liskov,1979] Barbara Liskov et al.: *Clu Reference Manual*. MIT/LCS/TR-225. MIT Cambridge, Mass. 1979.
- [Martin,1995] Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall. Englewood Cliffs, New Jersey. 1995. ISBN 0-13-203837-4.
- [Orwell,1949] George Orwell: *1984*. Secker and Warburg. London. 1949.
- [Parrington,1995] Graham Parrington et al.: *The Design and Implementation of Arjuna*. Computer Systems. Vol. 8 No. 3. Summer 1995.
- [Richards,1980] Martin Richards and Colin Whitby-Stevens: *BCPL – The Language and Its Compiler*. Cambridge University Press, Cambridge. England. 1980. ISBN 0-521-21965-5.
- [Rosler,1984] L. Rosler: *The Evolution of C – Past and Future*. AT&T Bell Laboratories Technical Journal. Vol. 63 No. 8. Part 2. October 1984.
- [Rozier,1988] M. Rozier, et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1 No. 4. Fall 1988.
- [Sethi,1981] Ravi Sethi: *Uniform Syntax for Type Expressions and Declarations*. Software Practice & Experience. Vol. 11. 1981.
- [Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). August, 1994.
- [Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Mass. 1986. ISBN 0-201-12078-X.
- [Stroustrup,1987] Bjarne Stroustrup and Jonathan Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ conference. Santa Fe, New Mexico. November 1987.
- [Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (Second Edition)*.

- Addison-Wesley. Reading, Mass. 1991. ISBN 0-201-53992-6.
- [Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.
- [Tarjan,1983] Robert E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, Penn. 1983. ISBN 0-898-71187-8.
- [Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley Developers Press. Reading, Mass. 1996. ISBN 0-201-48345-9.
- [UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition*. AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.
- [Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge. Mass. 1996. ISBN 0-262-73118-5.
- [Wikström,1987] Åke Wikström: *Functional Programming Using ML*. Prentice-Hall. Englewood Cliffs, New Jersey. 1987.
- [Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office. London. England. 1974.

关于设计和大型软件开发问题的参考文献和书籍，可以在第23章的最后找到。

第2章 C++ 概 览

我们要做的第一件事，就是杀掉所有的语言专家。

——《亨利六世》(第二部分)

为什么是C++——程序设计范型——过程式程序设计——模块化——分别编译——异常处理——数据抽象——用户定义类型——具体类型——抽象类型——虚函数——面向对象的程序设计——通用型程序设计——容器——算法——语言和程序设计——忠告

2.1 为什么是C++

C++是一种通用程序设计语言，特别是面向系统程序设计，它

- 是一个更好的C。
- 支持数据抽象。
- 支持面向对象的程序设计。
- 支持通用型程序设计。

本章将解释这些意味着什么，而又不涉足语言定义中更精微的细节。这样做的目的是给你一个有关C++以及使用它的关键性技术的一般性概貌，而不是为你提供开始做C++程序设计所必须的细节信息。

如果你发现本章的有某些部分跟不上，请尽管忽略它并继续前进，所有的东西在后面章节里都有详细的解释。当然，如果你真的跳过了本章中的某些部分，以后再返回来看看还是会很有帮助的。

对语言特征的细节理解——甚至有关一个语言的所有特征——也不能代替对该语言以及使用它的基本技术的全局性认识。

2.2 程序设计范型

面向对象的程序设计是一种程序设计技术——对一组问题写出“好”程序的一种范型。如果术语“面向对象的程序设计语言”有一点意思的话，它意味的就是某种程序设计语言特别提供了一些机制，以很好地支持在其中做面向对象风格的程序设计。

这里存在着一个重要的区分。说一个语言支持某种风格的程序设计，如果它提供了一些功能，使得它能够方便地（比较容易、安全和有效地）用于这种程序设计风格。如果要写那样的程序必须付出很大的努力或利用各种技巧，就说一个语言不支持某种技术。这样的语言只是允许使用这类技术。例如，你可以在Fortran77里写结构化程序，或在C里写面向对象的程序，但这样做时会出奇地困难，因为这些语言并不直接支持这些技术。

要支持一种范型，不仅在于某些能直接用于该种范型的显见形式的语言功能，还在于一些形式上更加细微的，对无意中偏离了这种范型的情况做编译时或者运行时的检查。类型检

查是这类事物中最明显的例子，歧义性检查和运行时检查也被用做对某种范型的语言支持。语言之外的功能，例如库和程序设计环境，也能进一步提供对一种范型的支持。

一种语言并不会因为拥有其他语言所没有的某种特征，就比其他的语言更好。这方面的反例太多了。最重要的问题并不在于某个语言究竟拥有多少特征，而在于它所拥有的特征是否足以在某个所希望的应用领域中支持某种所希望的程序设计风格：

- [1] 所有特征必须清晰而优美地集成在语言之中。
- [2] 必须能组合使用这些特征去得到一种解决方案，如果无法做这样组合、那就会要求额外的独立的特征。
- [3] 应尽可能减少荒谬的和“专用的”特征。
- [4] 任何特征的实现都不应该给未使用这种特征的程序强加明显的额外开销。
- [5] 用户只需要了解自己在写程序时所明确使用的那个语言子集。

第一条原则明显是美学的和逻辑的。随后两条表述的是最小化的思想。最后两条可以总结为“你不知道的东西不会伤害你”。

设计C++就是为了支持数据抽象、面向对象的程序设计和通用型程序设计，以及在这些风格约束之下的传统的C程序设计技术。它从未有意要给用户强加某种特殊的程序设计风格。

下面几节将考虑某些程序设计风格，以及支持它们的关键性语言机制。整个展示过程的进展是通过一系列技术，开始于过程式程序设计，而后引向在面向对象的程序设计中采用类层次结构，以及采用模板的通用型程序设计。每个范型都是在其前驱的基础上构造起来的，为C++程序员工具箱里加上一件新东西，也反映了一种已被证明非常有效的设计方法。

这里对语言特征的展示并不完全。在这里想强调的是设计方法以及组织程序的方式，而不是语言的细节。在这个阶段，把握什么东西可以用C++做的基本思想，远比正确地理解怎样才能做到它更重要得多。

2.3 过程式程序设计

原始的程序设计范型是：

确定你需要哪些过程；
采用你能找到的最好的算法。

这里所关注的是处理过程——执行预期的计算所需要的算法。支持这种范型的语言提供了一些功能，如给函数传递参数以及从函数返回结果值等。与这种思考方式相关的文献里充斥着与此有关的讨论：参数传递的各种方式，区分不同种类的参数或不同种类的函数（例如，过程，例行程序和宏等）的方式，如此等等。

“好风格”的一个典型实例是平方根函数。给定一个双精度浮点数的实际参数，该函数将产生出一个结果。为做此事，函数需要完成一段大家都已理解得很好的数学计算：

```
double sqrt(double arg)
{
    // 计算平方根的代码
}

void f()
```

```
{
    double root2 = sqrt(2),
    // ...
}
```

花括号 {} 在C++里表示结成组，在这里它们指明了函数体的开始和结束。由双斜线 // 开始的是一段直到行尾的注释。关键字 **void** 表明一个函数不返回值。

从程序组织的观点看，函数被人们用于在许多算法的迷宫里建立起一种秩序。算法本身通过函数调用和其他语言功能写出。下面几小节将简短地介绍C++中表达计算的各种最基本功能。

2.3.1 变量和算术

每个名字、每个表达式都有一个类型，以确定可以对它们执行的操作。例如，声明

```
int inch;
```

描述说 *inch* 的类型是 *int*，也就是说，*inch* 是一个整型变量。

一个声明是一个语句，它为程序引入一个名字，还为这个名字确定了一个类型。类型则定义了名字或者表达式的正确使用方式。

C++提供了一批各种各样的基本类型，它们都直接对应于一些硬件功能。例如

```
bool        // 布尔类型，可能的值是true和false
char        // 字符类型，例如 'a'、'z' 和 '9'
int         // 整数类型，例如1、42和1216
double      // 双精度浮点数类型，例如3.14和299793.0
```

一个 *char* 变量具有某种自然的大小，正好能保存给定机器里的一个字符（通常是一个字节），而一个 *int* 变量也具有某种自然的大小，正好适合给定机器里的整数算术（通常是一个机器字）。

算术运算可以用于这些类型的任意组合：

```
+          // 加，一元和二元
-          // 减，一元和二元
*          // 乘
/          // 除
%          // 余数
```

比较运算符也是这样：

```
==         // 等于
!=         // 不等于
<          // 小于
>          // 大于
<=         // 小于等于
>=         // 大于等于
```

在做赋值和算术运算时，C++能在基本类型之间完成所有有意义的相互转换，因此各种类型可以自由地混合使用：

```
void some_function()    // 函数不返回值
{
    double d = 2.2;      // 初始化浮点数
```

```

    int i = 7;           // 初始化整型变量
    d = d+i;            // 将和赋值给d
    i = d*i;            // 将乘积赋值给i
}

```

像在C中一样，`=` 是赋值符号，而 `==` 检测相等。

2.3.2 检测和循环

C++提供了一组很方便的用于表示选择和循环的语句。举例来说，这里是一个简单的函数，它提示用户输入，并返回一个布尔值代表这个输入：

```

bool accept()
{
    cout << "Do you want to proceed (y or n)?\n"; // 写出提问

    char answer = 0;
    cin >> answer;                               // 读回答

    if (answer == 'y') return true;
    return false;
}

```

运算符 `<<` (“送出”) 用做输出运算符；`cout` 是标准输出流。运算符 `>>` (“取入”) 用做输入运算符；`cin` 是标准输入流。`>>` 的右运算对象的类型确定能接受什么样的输入，它也作为输入操作的目标。在输出字符串最后的 `\n` 字符表示换行。

通过将回答‘`n`’也纳入考虑的范围，可以对这个例子做一点改进：

```

bool accept2()
{
    cout << "Do you want to proceed (y or n)?\n"; // 写出提问

    char answer = 0;
    cin >> answer;                               // 读回答

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "I'll take that for a no.\n";
        return false;
    }
}

```

这里的`switch`语句对照一组常量检查一个值，这些分情况常量必须互不相同。而如果这个值与所有常量都不匹配，那么就选择`default`。程序员可以不提供`default`。

很少有程序里不包含循环。对于目前情况，我们可能希望允许用户试几次：

```

bool accept3()
{
    int tries = 1;
    while (tries < 4) {
        cout << "Do you want to proceed (y or n)?\n"; // 写出提问
        char answer = 0;
    }
}

```

```

    cin >> answer;                                // 读回答

    switch (answer) {
    case 'y':
        return true;
    case 'n':
        return false;
    default:
        cout << "Sorry, I don't understand that.\n";
        tries = tries + 1;
    }
}
cout << "I'll take that for a no.\n";
return false;
}

```

这里的while语句一直执行到条件变成false为止。

2.3.3 指针和数组

数组可以如下的定义：

```
char v[10];    // 10个字符的数组
```

指针的定义与此类似

```
char* p;    // 指向字符的指针
```

在声明里，[] 表示“的数组”，而 * 表示“的指针”。所有数组都以0作为它们的下界。因此v有10个元素，v[0]... v[9]。指针可以保有适当类型的对象的地址：

```
p = &v[3];    // p指向v的第4个元素
```

一元的 & 是取地址运算符。

考虑将10个元素从一个数组复制到另一个数组：

```

void another function()
{
    int v1[10];
    int v2[10];
    // ...
    for (int i=0; i<10; ++i) v1[i]=v2[i];
}

```

这里的for语句可以读作“将i设置为0，当i小于10时，做第i个元素的赋值并增大i的值”。在应用到整型变量上时，增量运算符 ++ 简单地将变量的值加1。

2.4 模块程序设计

在这些年里，设计程序的着重点已经从有关过程的设计转移到对数据的组织了。除了其他因素之外，这种转移也反映了程序规模增大的情况。一集相关的过程与被它们所操作的数据组织在一起，通常被称做一个模块。程序设计的范型变成了：

确定你需要哪些模块；将程序分为
一些模块，使数据隐藏于模块之中。

这一范型也被作为数据隐藏原理而广为人知。在那些不存在与数据相关的过程组之处，采用过程式程序设计也就足够了。还有，设计“好过程”的技术现在可以应用到模块中的各个过程。有关模块的最常见的例子是定义一个堆栈。这里需要解决的主要问题是：

- [1] 为堆栈提供一个用户界面（例如，函数`push()`和`pop()`）。
- [2] 保证堆栈的表示（例如，一个元素的数组）只能通过用户界面访问。
- [3] 保证堆栈在被使用之前已经做了初始化。

C++提供了一种机制，可以把相关的数据、函数等组织到一个独立的名字空间里。例如，模块`Stack`的用户界面可以按如下方式声明和使用：

```
namespace Stack {           // 界面
    void push(char);
    char pop();
}

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

这里的`Stack::`限定词表明`push()`和`pop()`是来自`Stack`名字空间。这些名字的其他使用将不会与之干扰，不会引起混乱。

`Stack`的定义可以通过程序的另一个单独编译的部分提供：

```
namespace Stack {           // 实现
    const int max_size = 200;
    char v[max_size];
    int top = 0;

    void push(char c) { /* 检查上溢并压入c */ }
    char pop() { /* 检查下溢并弹出 */ }
}
```

有关这个`Stack`模块的关键点是，用户代码完全被隔离于`Stack`的数据表示之外，隔离的方式是通过写出`Stack::push()`和`Stack::pop()`代码来实现。用户不必知道`Stack`是用数组实现的，这个实现方式也可以修改，而不会影响用户的代码。这里以 `/*` 开始一段注释，以 `*/` 结束注释。

数据实际上只是人们希望“隐藏”起来的许多东西中的一类，数据隐藏的概念很容易扩展到信息隐藏，也就是说，例如函数、类型等的名字，也很容易做成一个模块里面局部的东西。为此，C++允许把任何声明放到名字空间里（8.2节）。

这个`Stack`模块只是描述堆栈的一种方式，下面几小节将用各种堆栈作为例子，阐释不同的程序设计风格。

2.4.1 分别编译

C++支持C语言中有关分别编译的概念。这种机制可以用于将程序组织为一组部分独立的片段。

典型地，我们将描述一个模块的界面的声明放进一个文件里，以文件名表示它的正当使用方式。这样，

```
namespace Stack {           // 界面
    void push(char);
    char pop();
}
```

可能被放入文件`stack.h`，堆栈的用户将像下面这样包含该文件（这种文件称为头文件）：

```
#include "stack.h"           // 获得界面

void f()
{
    Stack::push('c');
    if (Stack::pop() != 'c') error("impossible");
}
```

为了帮助编译器保证程序的一致性，提供了`Stack`模块实现的文件也要包含这个文件：

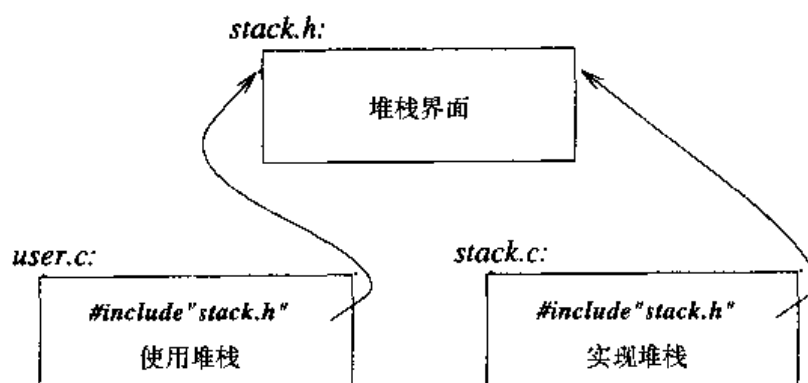
```
#include "stack.h"           // 获得界面

namespace Stack {           // 表示
    const int max_size = 200;
    char v[max_size];
    int top = 0;
}

void Stack::push(char c) { /* 检查上溢并压入c */ }

char Stack::pop() { /* 检查下溢并弹出 */ }
```

用户代码放在第三个文件里，例如位于`user.c`中。在`stack.c`和`user.c`中的代码共享由`stack.h`提供的堆栈界面信息，除此之外这两个文件就是互不相关的，可以分别进行编译。这些程序片段可以用下面的图表示：



分别编译是一切实际程序中都需要考虑的问题。在程序里，不能简单地将所提供的功能，例如`Stack`，当做模块。严格地说，分别编译的使用并不是语言要考虑的问题，而是关于如何最好地利用特定语言实现的优点的问题。当然，在实践中这个问题是极其重要的。最好的方式就是最大限度地模块化，通过语言特征去逻辑地表示模块化，而后通过能最有效地分别编译的一组文件，物理地利用这种模块化机制（第8、9章）。

2.4.2 异常处理

当一个程序被设计为一集模块以后，对于错误的处理也必须在这些模块的基础之上考虑。哪个模块应该承担对于哪个错误的处理责任？常常遇到的情况是，检查出错误的模块并不知

道应该去做些什么，恢复的动作依赖于那些调用操作的模块，而不是那个试图去执行操作并且发现了错误的模块。随着程序不断增大，特别是随着各种库的广泛使用，处理错误（或者更一般的，“异常情形”）的标准方式也变得更加重要了。

重新考虑有关`Stack`的例子。如果我们试图向堆栈中`push()`放进的字符过多，这时应该怎么办呢？写`Stack`的人不可能知道在这种情况下用户希望做些什么，而用户也未必能够始终如一地检查这个问题（如果真能这样做的话，溢出问题根本就不会出现了）。解决的办法就是让`Stack`的实现者去检查溢出问题，而后告诉那个（它并不知道是什么的）用户。这样就使用户可以采取适当的行动。例如，

```
namespace Stack {           // 界面
    void push(char);
    char pop();

    class Overflow {}; // 表示溢出异常的类型
}
```

当检查到溢出之时，`Stack::push()`可以激活异常处理代码；也就是说，它“抛出一个`Overflow`异常”：

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    // 压入c
}
```

这个`throw`将把控制传递给某个能处理`Stack::Overflow`异常的处理器，该处理器应该位于某个直接或者间接调用了`Stack::push()`的函数里。在这个传递过程中，实现将根据需要退掉一部分函数调用栈，退回到能处理异常的那个调用函数的环境位置。可见，抛出的行为就像是一种多层的返回。例如，

```
void f()
{
    // ...
    try { // 这里的异常将由下面定义的处理器处理
        while (true) Stack::push('c');
    }
    catch (Stack::Overflow) {
        // 呜呼，出现异常；执行适当动作
    }
    // ...
}
```

这里的`while`循环将不断地做下去。当然，在某个对`Stack::push()`的调用导致`throw`时，执行就将进入提供了针对`Stack::Overflow`的处理器器的`catch`子句之中。

采用异常处理机制能使对错误的处理更加规范，也使它更容易看清楚。参看8.3节、第14章和附录E中的进一步讨论、细节问题和程序实例。

2.5 数据抽象

模块化是一切成功的大型程序的一个最基本特征。它也将是贯穿本书中所有有关设计的

讨论的一个中心。然而，像前面所描述的那种模块形式，对于清晰地表示复杂的系统而言还是不够的。在这里，我将先展示一种使用模块的方式，用以提供一种用户定义类型的形式，而后再说明如何通过直接定义用户定义类型，以便克服此方式中的各种缺陷。

2.5.1 定义类型的模块

基于模块的程序设计趋向于以一个类型的所有数据为中心，在某个类型管理模块的控制之下工作。例如，如果我们希望有许多堆栈——而不是像前面那样只用一个由`Stack`模块提供的堆栈——我们就可能会定义一个堆栈管理器，它具有如下的界面：

```
namespace Stack {
    struct Rep;           // 在另外某个地方定义堆栈的布局
    typedef Rep& stack;

    stack create();        // 做出一个新堆栈
    void destroy(stack s); // 删除s

    void push(stack s, char c); // 将c压入s
    char pop(stack s);        // 弹出s
}
```

声明

```
struct Rep;
```

说`Rep`是一个类型的名字，但将这个类型留待以后再去定义（5.7节）。声明

```
typedef Rep& stack;
```

确定名字`stack`是“对`Rep`的引用”（细节见5.5节）。这里的想法是，堆栈由`Stack::stack`表示，而进一步的细节则对用户隐藏起来。

一个`Stack::stack`用起来很像一个内部类型的变量：

```
struct Bad_pop { };

void f()
{
    Stack::stack s1 = Stack::create(); // 做出一个新堆栈
    Stack::stack s2 = Stack::create(); // 做出另一个新堆栈

    Stack::push(s1, 'c');
    Stack::push(s2, 'k');

    if (Stack::pop(s1) != 'c') throw Bad_pop();
    if (Stack::pop(s2) != 'k') throw Bad_pop();

    Stack::destroy(s1);
    Stack::destroy(s2);
}
```

我们可以以几种不同的方式来实现这个堆栈。最重要的是，用户完全不必知道我们到底是怎么做的。只要我们能保持这个界面不改变，即使是决定重新实现`Stack`，用户也不会受到任何影响。

某个实现可以是预先分配几个堆栈，而让`Stack::create()`递交出到某个尚未使用的堆栈的引用。而`Stack::destroy()`则将一个堆栈表示标记为未使用的，以使`Stack::create()`能够重新用它


```

namespace Stack { // 表示
    const int max_size = 200;

    struct Rep {
        char v[max_size];
        int top;
    };

    const int max = 16; // 最大堆栈数

    Rep stacks[max]; // 预分配的堆栈表示
    bool used[max]; // 如果stacks[i]在使用, 则used[i]为真

    typedef Rep& stack;
}

void Stack::push(stack s, char c) { /* 检查s的上溢并压入c */ }

char Stack::pop(stack s) { /* 检查s的下溢并弹出 */ }

Stack::stack Stack::create()
{
    // 找一个未使用的Rep, 将它标记为已使用的, 将它初始化, 并返回它的引用
}

void Stack::destroy(stack s) { /* 标记s为未使用的 */ }

```

我们在这里做的就是围绕着一个表示类型包装起一组界面函数。所做的结果得到的“堆栈类型”具有怎样的行为，部分在于我们如何定义这些界面函数，部分在于我们如何将`Stack`的表示展示给它的用户，部分在于这个表示本身的设计情况。

这种做法常常不是最理想的。一个重要问题就是，这里给用户提供了一个“假类型”的表示，它可以因为表示类型不同而出现很大变化——而与此同时，又应该将用户隔离于这一表示之外。例如，如果我们选择采用另外一种更精致的数据结构去表示一个堆栈，那么`Stack::stack`的初始化和赋值规则就可能发生很剧烈的变化。而这种事情确实是常常需要做的。这种情况实际上说明了，我们在这里只是简单地把提供方便的堆栈的问题从`Stack`模块移到了`Stack::stack`的表示类型。

更根本的问题是，通过模块实现的这种用户定义类型，它们所提供的对这种类型的访问，在行为上，并不像内部的类型。它们得到的支持也与内部类型不同，而且实际上更少一些。例如，一个`Stack::Rep`能被使用的期间由`Stack::create()`和`Stack::destroy()`控制，而不是由普遍性的语法规则控制。

2.5.2 用户定义类型

C++ 解决这个问题方式就是允许用户直接定义类型，这种类型的行为方式（几乎）与内部类型完全一样。这样的类型常常被称做抽象数据类型，而我更喜欢术语用户定义类型。有关抽象数据类型的一个更合理的定义应当要求一个数学的“抽象”描述。给出了一个这样的描述之后，我们这里所谓的类型就是那种真正抽象实体的一个具体实例。现在程序设计范型变成了：

确定你需要哪些类型；
为每个类型提供完整的一组操作。

在那些对于每个类型都只需要一个对象的地方，采用模块方式实现数据隐藏风格的程序设计也就足够了。

各种算术类型，例如有理数或复数，是用户定义类型的最常见实例。例如，

```
class complex {
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; } // 从两个标量构造一个复数
    complex(double r) { re=r; im=0; } // 从一个标量构造一个复数
    complex() { re = im = 0; } // 默认的复数: (0, 0)

    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex); // 二元
    friend complex operator-(complex); // 一元
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);

    friend bool operator==(complex, complex); // 等于
    friend bool operator!=(complex, complex); // 不等于
    // ...
};
```

类（也就是用户定义类型）**complex**的声明描述了一个复数的表示，以及一组对复数的操作。这个表示是私用的，也就是说，**re**和**im**只能由类**complex**的声明里描述的那些函数访问。有关的函数可以按如下方式定义：

```
complex operator+(complex a1, complex a2)
{
    return complex(a1.re+a2.re, a1.im+a2.im);
}
```

名字与类名相同的成员函数被称为构造函数。每个构造函数定义了一种初始化这个类的对象的方式。类**complex**提供了三个构造函数，其中一个从一个**double**做出一个**complex**，另一个从一对**double**做出一个**complex**，第三个做出一个具有默认值的**complex**。

类**complex**可以按如下方式使用：

```
void f(complex z)
{
    complex a = 2.3;
    complex b = 1/a;
    complex c = a+b*complex(1, 2.3);
    // ...
    if (c != b) c = -(b/a)+2*b;
}
```

编译器将把涉及到**complex**数的各种运算转变为适当的函数调用。例如，**c != b**的意思是 **operator != (c, b)**，而 **1/a** 的意思是 **operator / (complex(1), a)**。

大部分（但并不是全部）模块表示为用户定义类型更好一些。

2.5.3 具体类型

可以设计出许多用户定义类型去满足各种各样的需要。现在来考虑一个遵循**complex**那种定义方式的用户定义的**Stack**类型。为了使这个例子更实际一点，这个**Stack**类型的定义以它的

元素个数作为一个参数:

```
class Stack {
    char* v;
    int top;
    int max_size;
public:
    class Underflow { };    // 用做异常
    class Overflow { };    // 用做异常
    class Bad_size { };    // 用做异常

    Stack(int s);           // 构造函数
    ~Stack();               // 析构函数

    void push(char c);
    char pop();
};
```

构造函数`Stack(int)`将在建立这个类的对象时被调用,它处理初始化问题。如果该类的一个对象出了其作用域,需要做某些清理时,就应该去声明构造函数的对应物——它被称为析构函数:

```
Stack::Stack(int s)        // 构造函数
{
    top = 0;
    if (s < 0 || 10000 < s) throw Bad_size();    // “||”的意思是“或”
    max_size = s;
    v = new char[s];    // 在自由存储(堆,动态存储)中为元素分配存储
}

Stack::~~Stack()           // 析构函数
{
    delete[] v;            // 释放元素存储,使空间可能重新使用(6.2.6节)
}
```

构造函数初始化新的`Stack`变量,在做这件事时,它用`new`运算符从自由空间(也称为堆或动态存储)分配一些存储。析构函数进行清理,方式就是释放这些存储。所有这些都能在无需`Stack`的用户干预的情况下完成。有关用户只要简单地建立和使用`Stack`,就像它们是内部类型的变量似的。例如,

```
Stack s_var1(10);          // 具有10个元素的全局堆栈

void f(Stack& s_ref, int i) // 对Stack的引用
{
    Stack s_var2(i);        // 具有i个元素的局部堆栈
    Stack* s_ptr = new Stack(20); // 指向在自由存储分配的Stack

    s_var1.push('a');
    s_var2.push('b');
    s_ref.push('c');
    s_ptr->push('d');
    // ...
}
```

这个`Stack`类型遵循与内部类型(例如`int`和`char`等)同样的规则,包括命名、作用域、存储分配、生存时间、复制等。

当然，成员函数`push()`和`pop()`也必须在某个地方给予定义：

```
void Stack::push(char c)
{
    if (top == max_size) throw Overflow();
    v[top] = c;
    top = top + 1;
}

char Stack::pop()
{
    if (top == 0) throw Underflow();
    top = top - 1;
    return v[top];
}
```

像`complex`和`Stack`这样的类型称为具体类型，它们与抽象类型相对应。抽象类型的界面能更完全地将用户与实现细节隔离。

2.5.4 抽象类型

在将`Stack`从用一个模块实现的“假类型”（2.5.1节）转变为一个真正的类型（2.5.3节）的过程中，有一个性质被丢掉了：表示方式没有与用户界面分离，反而变成了使用`Stack`的程序片段里将要包含的一个部分。这个表示完全是私用的，因此只能通过成员函数访问，然而它却出现在那里。如果这个表示有了某种显著变化，那些使用它的代码就必须重新编译。这是为做出在行为方式上完全像内部类型的具体类型时所付出的一个代价。特别因为在不知道一个类型表示的大小的情况下，我们就无法获得这个类型的真正的局部变量。

在类型不常改变，而且局部变量又确实提供了我们极需要的清晰性和效率的那些地方，这种方式是完全可以接受的，也是很理想的。但是，如果我们需要将堆栈的用户与堆栈表示的修改完全隔离开，前面的这个`Stack`就不够好了。针对这个问题的解决方案能得到界面与表示的完全分离，这时需要放弃的就是真正的局部变量。

我们首先定义界面：

```
class Stack {
public:
    class Underflow { };    // 用于异常
    class Overflow { };     // 用于异常

    virtual void push(char c) = 0;
    virtual char pop() = 0;
};
```

词语`virtual`在Simula和C++里的意思是“可以在今后由这个类所派生的类里重新定义”。由`Stack`派生出的一个类将提供这个`Stack`界面的一个具体实现。有些古怪的`=0`语法形式说明在由`Stack`派生的某些类中必须定义这些函数。这样，这个`Stack`就能作为任何实现`push()`和`pop()`的类的界面了。

该`Stack`以如下方式使用：

```
void f(Stack& s_ref)
{
    s_ref.push('c');
```

```

        if (s_ref.pop() != 'c') throw Bad_pop();
    }

```

请注意`f()`如何使用着`Stack`界面，而完全忽略了实现的细节。为另外一些不同的类提供界面的类也常常被称做多态类型。

毫不奇怪，有关实现的所有东西都可以与前面那个具体`Stack`类完全一样，我们就是从那里提取出了界面`Stack`：

```

class Array_stack : public Stack {    // Array_stack实现Stack
    char* p;
    int max_size;
    int top;
public:
    Array_stack(int s);
    ~Array_stack();

    void push(char c);
    char pop();
};

```

开始处的“`public`”可以读作“由其派生”，“实现”或者“是它的子类型”。

对于像`f()`这样的使用着`Stack`，但却完全不管其实现细节的函数，需要有另外的函数去为它创建对象，使`f()`一类的函数可以在这些对象上操作。例如，

```

void g()
{
    Array_stack as(200);
    f(as);
}

```

因为`f()`根本不知道`Array_stack`，只知道`Stack`界面，它在`Stack`的另一个不同实现上工作起来也同样好。例如，

```

class List_stack : public Stack {    // List_stack实现Stack
    list<char> lc;                    // (标准库) 字符的表 (3.7.3节)
public:
    List_stack() {}

    void push(char c) { lc.push_front(c); }
    char pop();
};

char List_stack::pop()
{
    char x = lc.front();              // 取得第一个元素
    lc.pop_front();                   // 删除第一个元素
    return x;
}

```

在这里，具体表示采用了一个字符的表。`lc.push_front(c)`将`c`添加为`lc`的第一个字符，调用`lc.pop_front()`删除第一个字符，而`lc.front()`表示`lc`的第一个字符。

下面函数建立起一个`List_stack`，并让`f()`去使用它：

```

void h()
{
    List_stack ls;
}

```

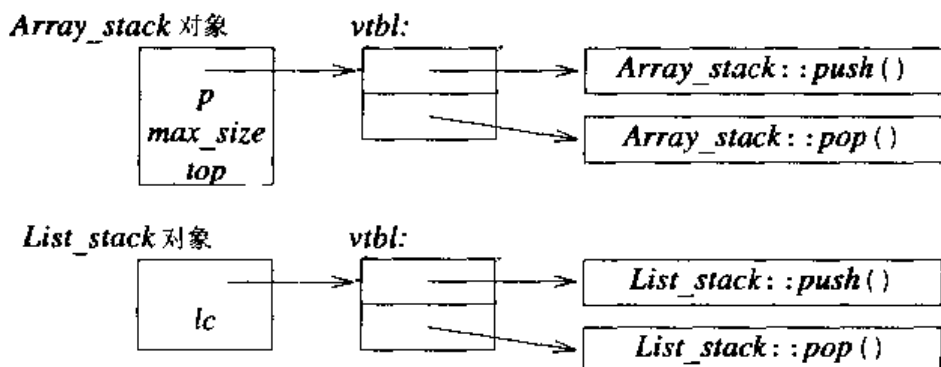
```

    f(ls);
}

```

2.5.5 虚函数

怎样才能将 $f()$ 里对 $s_ref.pop()$ 的调用解析为正确的函数定义呢? 当 $f()$ 被 $h()$ 调用时, 实际上必须去调用 $List_stack::pop()$ 。为能得到这种解析, 在 $Stack$ 对象里就必须包含某些信息, 指明在程序运行中应该调用的函数。一种常用的实现技术是让编译器把一个 $virtual$ 函数的名字转换为指向这些函数的指针表的一个下标。这种表通常被称为“虚函数表”, 或简称 $vtbl$ 。每个带有虚函数的类都有标识着它的所有虚函数的 $vtbl$, 这个情况可以用下图表示:



即使调用者并不知道对象的大小以及它的数据布局, 位于 $vtbl$ 里的函数也使对象能正确地使用。调用者需要知道所有东西就是 $Stack$ 的那个 $vtbl$ 的位置, 以及对各个虚函数应该使用的下标。这种虚函数调用机制的效率可以做得基本上与“正常函数调用”机制相同。其空间开销是带有虚函数的类的每个对象里包含一个指针, 而每个这样的类需要有一个 $vtbl$ 。

2.6 面向对象的程序设计

对于好的设计而言, 数据抽象是最基本的东西, 它也将是贯穿本书讨论设计时所特别关注的问题。但是, 只有用户定义类型对我们的需要而言还不够灵活。本节将首先说明简单的用户定义类型中的一个问题, 而后展示如何利用类层次结构的方式去解决它。

2.6.1 具体类型的问题

一个具体类型就像通过模块定义的一个“假类型”一样, 它定义了一类黑盒子。一旦这种黑盒子定义好之后, 它也就无法再与程序的其他部分进行实际地交互: 没有任何方式可以为了某些新的用途而调整它, 除非去修改它的定义。这种情况可以认为是非常理想的, 但也会导致严重缺乏灵活性。考虑在某个图形系统里定义一个类型 $Shape$ 。假定当时这个系统需要支持圆形、三角形和正方形, 再假定我们已经有了

```

class Point{ /* ... */ };
class Color{ /* ... */ };

```

这里的 $/*$ 和 $*/$ 分别表示一个注释的开始和结束, 这种注释形式可以用于多行的注释, 以及在一行结束之前结束的那种注释。

我们可能像下面这样定义形状($shape$)类:

```

enum Kind { circle, triangle, square }; // 枚举 (4.8节)

class Shape {
    Kind k;           // 类型域
    Point center;
    Color col;
    // ...

public:
    void draw();
    void rotate(int);
    // ...
};

```

这里的“类型域”*k*是必需的,以便使`draw()`和`rotate()`一类的函数能够确定它们正在处理的是哪种形状(在类Pascal的语言里,人们可以使用带有标志*k*的变体记录)。函数`draw()`的定义可能具有下面的样子:

```

void Shape::draw()
{
    switch (k) {
        case circle:
            // 画圆
            break;
        case triangle:
            // 画三角形
            break;
        case square:
            // 画正方形
            break;
    }
}

```

这真是一个烂泥潭,像`draw()`这一类函数“必须知道”现存的所有形状的种类。因此,每当有一种新形状被加进系统时,这类函数中每一个的代码都需要增加。如果我们定义了一种新形状,就需要检查每个有关形状的操作并(可能)做些修改。除非我们能使用所有操作的源代码,否则我们就无法向这个系统中加进一个新形状。由于增加一个新形状将涉及到“触动”在形状上的每个重要操作的代码,这样做的时候就需要极高的技术水平,也很可能将错误引进处理其他(老)形状的代码中。还有,对于特定形状的表达方式的选择也受到严重的束缚,因为它们的表示(至少是某些部分)必须塞进由通用类型`Shape`的定义所表达的常常是具有固定大小的框架之中。

2.6.2 类层次结构

这里的问题在于:没有一种方式来区分所有形状的共有特征(例如,一个形状总有一种颜色、可以被画出来等),与某个特定形状种类的特殊特征(如圆是一种形状,它有一个半径,需要用画圆的函数画出等)。表达这种区分并由此获益就定义了面向对象的程序设计。具有能表达和利用这种区分的结构的语言将能支持面向对象的程序设计,其他语言就不行。

继承机制(是C++从Simula借来的)提供了一种解决方案。首先,我们描述一个定义了所有形状的共同特征的类:

```

class Shape {
    Point center;
    Color col;
    // ...
public:
    Point where() { return center; }
    void move(Point to) { center = to; /* ... */ draw(); }

    virtual void draw() = 0;
    virtual void rotate(int angle) = 0;
    // ...
};

```

就像在2.5.4节中的抽象类型`Stack`，所有提供了被定义的调用界面——但其实现却不能定义——的函数都是`virtual`。特别地，函数`draw()`和`rotate()`只能针对特定形状的语义去定义，所以它们被声明为`virtual`。

在给出这个定义之后，我们就可以写出函数，实现某种针对指向形状的指针向量的操作：

```

void rotate_all(vector<Shape*>& v, int angle) // 将v的所有元素旋转angle度
{
    for (int i = 0; i < v.size(); ++i) v[i] -> rotate(angle);
}

```

要定义出一个特殊的形状，我们必须说明它是一个形状，并描述它的特殊性质（包括那些虚函数）：

```

class Circle : public Shape {
    int radius;
public:
    void draw() { /* ... */ }
    void rotate(int) {} // 的确，这是个空函数
};

```

在C++里，类`Circle`被称为是由类`Shape`派生的，而类`Shape`被称为是类`Circle`的基类。关于`Circle`和`Shape`的另一对术语是子类和超类。我们还说派生类由其基类继承了成员，所以，对基类和派生类的使用通常也被说成是继承。

现在程序设计范型是：

确定你需要哪些类；
为每个类提供完整的一组操作；
利用继承去明确地表示共性。

在不存在共性的地方，数据抽象就足够了。在类型之间，能够通过使用继承和虚函数挖掘出的共性的数量，可以看做是面向对象程序设计对于该问题的适用性的一个石蕊^①检验。在一些领域中，例如交互式图形，存在着面向对象程序设计的广阔空间。而在另一些领域，例如经典的算术类型和基于它们的计算领域，看来就很少有什么地方需要数据抽象之外的东西了，为支持面向对象程序设计而提供的功能在这里就未必是必需的。

① 在碱性溶液中呈蓝色，在酸性溶液中呈红色，广泛用做指示剂。——译者注

在一个系统中找出共性并不是一件不值一提的事情。能够利用的共性的量也受到系统设计方式的影响。在设计系统的时候——甚至是在写有关系统的需求时——就应该主动地去考虑共性问题。可以特别地将类设计为构造其他类型的基本构件。现存的类也应该检查，看看它们是否表现出某些共性，能够通过一个基类加以利用。

要想去弄清楚什么是面向对象的程序设计，而又不依赖于特定的程序设计语言结构，可以参看23.6节中给出的 [Kerr, 1987] 和 [Booch, 1994]。

类层次结构和抽象类（2.5.4节）互为补充而不是互相排斥（12.5节）。一般来说，列在这里的所有范型都是互相补充的，常常是互相支持的。例如，类和模块都包含着函数，而模块又包含着类和函数。有经验的程序员会根据需要去选择使用各种不同的范型。

2.7 通用型程序设计

有人需要用堆栈，但未必总是需要字符的堆栈。堆栈是一个具有一般性的概念，与字符的概念并无关系。因此，它应该能够被独立地表达。

更一般的情况是，如果一个算法能以独立于其表示细节的方式表达，而如果这样做又是能负担得起的，不出现逻辑毛病的话，那么就应该这样去做。

这个程序设计范型是：

确定你需要哪些算法；
将它们参数化，使它们能够对
各种各样适当的类型和数据结构工作。

2.7.1 容器

我们可以将字符堆栈类型推广到一个任意类型的堆栈的类型，方法是将它做成一个模板（*template*），用一个模板参数取代特定的类型*char*。例如，

```
template<class T> class Stack {
    T* v;
    int max_size;
    int top;
public:
    class Underflow { };
    class Overflow { };

    Stack(int s); // 构造函数
    ~Stack();     // 析构函数

    void push(T);
    T pop();
};
```

前缀`template<class T>`说明在以此作为前缀的声明中，*T*将被看做是一个参数。

成员函数可以类似地定义为：

```
template<class T> void Stack<T>::push(T c)
{
```

```

        if (top == max_size) throw Overflow();
        v[top] = c;
        top = top + 1;
    }
    template<class T> T Stack<T>::pop()
    {
        if (top == 0) throw Underflow();
        top = top - 1;
        return v[top];
    }

```

有了这些定义之后，我们就可以按

```

Stack<char> sc(200);           // 200个字符的堆栈
Stack<complex> scplx(30);      // 30个复数的堆栈
Stack<list<int>> sli(45);      // 45个整数表的堆栈

void f()
{
    sc.push('c');
    if (sc.pop() != 'c') throw Bad_pop();

    scplx.push(complex(1,2));
    if (scplx.pop() != complex(1,2)) throw Bad_pop();
}

```

方式使用堆栈了。类似地，我们也能将表、向量、映射（也就是关联数组）等都定义为模板。一个能保存某种类型的一集元素的类，一般被称为一个容器类，或简单地称做容器。

模板是一种编译时的机制，因此，与“手工编写的代码”相比，它们的使用并不引起任何额外的运行时开销。

2.7.2 通用型算法

C++标准库提供了各种各样的容器，用户也可以自己写出这类东西（第3、17、18章）。由此我们发现，可以进一步利用通用型程序设计范型，用容器对算法进行参数化。举例来说，我们希望能去排序、复制、检索 *vector*、*list* 或者数组，而不必对每种容器写出一套特殊的 *sort()*、*copy()* 和 *search()* 函数。我们当然也不希望将它们都转换到某个排序函数能够接受的特定类型。因此，我们就必须找到一种一般性的方式来定义自己的容器类，以使我们能操纵一个容器，而又不必确切地知道它到底是哪种容器。

有一种方式，也是C++标准库（3.8节、第18章）的容器类和非数值算法所采用的方式，是将注意力集中到序列和通过迭代器操作序列的观念。

下面是序列概念的一个图示



一个序列有一个开始和一个结束。一个迭代器引用着一个元素，并提供一种操作，通过它可以使迭代器转而去引用序列中的下一个元素。序列结束也是一个迭代器，它引用的是超出序列最后元素一个位置的地方。这种“结束”的物理表示可能是一个特殊的哨兵元素，但也不

一定非要这样做。事实上，最重要的论点是，这种序列的概念包容了范围广泛多样的许多表示形式，包括表和数组等。

我们需要一些标准的操作概念，例如“通过迭代器去访问元素”和“让这个迭代器去引用下一个元素”等。一种明显的选择（如果你理解这里的想法）是用间接运算符 `*` 表示“通过迭代器访问元素”，用增量运算符 `++` 表示“让这个迭代器去引用下一个元素”。

有了这些，我们就可以写出类似下面这样的代码：

```
template<class In, class Out> void copy(In from, In too_far, Out to)
{
    while (from != too_far) {
        *to = *from;    // 复制被索引的元素
        ++to;           // 下一个输出
        ++from;         // 下一个输入
    }
}
```

这样就可以复制任何容器，只要我们按正确的语法和语义为之定义了迭代器。

C++ 内部的、低级的数组和指针类型也有对应于这些的正确操作，因此我们可以写出如下代码：

```
char vc1[200]; // 200个字符的数组
char vc2[500]; // 500个字符的数组

void f()
{
    copy(&vc1[0], &vc1[200], &vc2[0]);
}
```

这将把 `vc1` 中从第一个到最后一个的元素复制到 `vc2`，从 `vc2` 的第一个元素开始放。

所有的标准库容器（16.3节、第17章）都支持这种迭代器和序列的概念。

这里用了两个模板参数 `In` 和 `Out` 表示复制的源和目标的类型，而不是只用一个参数。这是因为我们常常希望从一类容器复制到另一类容器。例如，

```
complex ac[200];

void g(vector<complex>& vc, list<complex>& lc)
{
    copy(&ac[0], &ac[200], lc.begin());
    copy(lc.begin(), lc.end(), vc.begin());
}
```

这里从数组复制到表，而后又从表复制到 `vector`。对于标准容器而言，`begin()` 就是一个迭代器，它指向容器的第一个元素。

2.8 附言

没有一种程序设计语言是完美无缺的。幸运的是，一种程序设计语言不必是完美无缺的，也可以成为构造伟大系统的良好工具。事实上，一种通用的程序设计语言根本不可能对它被用于的所有工作都是最完美的，对一项工作来说最完美的东西对于另外的工作就常常会表现出严重的缺陷，因为在一个领域中的完美事物实际上也就意味着专门化。C++ 的设计是想成为构造范围广泛多样的系统的良好工具，而且能够直接表达范围广泛多样的思想。

并不是所有东西都能直接通过语言的某些种内部特征表述。事实上，这也不应该成为一种理想。语言特征的存在是为了支持各种各样的程序设计风格和技术。因此，学习一种语言的工作就应该集中于去把握对该语言而言固有的和自然的风格——而不是去理解该语言的所有语言特征的细枝末节。

在实践性的程序设计中，理解语言中最晦涩难懂的语言特征，或者使用最大量的不同特征并不能获得什么利益。把一种特征孤立起来看并没有什么意思，只是在由技术和其他特征所形成的环境里，这一特征才获得了意义和趣味。因此，在阅读后面的章节时请牢记，考察C++的各种细节的真实目的在于能够应用它们，在有效设计的环境里，去支持良好的程序设计风格。

2.9 忠告

- [1] 不用害怕，一切都会随着时间的推移而逐渐明朗起来；2.1节。
- [2] 你并不需要在知道了C++的所有细节之后才能写出好的C++程序；1.7节。
- [3] 请特别关注程序设计技术，而不是各种语言特征；2.1节。

第3章 标准库概览

要是即刻就忘，何必费时去学？

——Hobbes

标准库——输出——字符串——输入——向量——范围检查——表——映射——容器概览——算法——迭代器——I/O迭代器——遍历和谓词——使用成员函数的算法——算法概览——复数——向量算术——标准库概览——忠告

3.1 引言

没有任何一个重要程序是只用某种赤裸裸的程序设计语言写出的。首先总是要开发出一组支撑库，这也就形成了进一步工作的基础。

继续第2章的讨论，本章将对关键性的库功能做一个快速的浏览，以便使你得到一个有关用C++及其标准库能够做什么的初步认识。这里将介绍各种有用的库类型，如`string`、`vector`、`list`和`map`等，同时将展示使用它们的最普通方式。这样做，就使我们能在随后的章节里给出一些更好的实例，以及一些更好的练习题。如在第2章中那样，这里还是要特别建议你，无须为自己不能完全理解各种细节而忧虑或沮丧。本章的目的不过是想给你关于将会遇到些什么的一种体验，并传递一些对最有用的库功能的最简单使用的认识。对标准库的详细介绍见16.1.2节。

在本书中描述的标准库功能只是每个完整的C++实现中的一部分。除了标准的C++库之外，大部分实现都提供了为使用者与程序交互服务的“图形用户界面”系统，它常被说成是GUI或者窗口系统。与此类似，大部分应用开发环境还提供了一些“基础库”，以支持公司或者业界的“标准”开发过程或/和执行环境。我将不去描述那些系统或者库，这里的意图是按照C++语言的标准定义，提供一个自足的描述，并保持所有实例的可移植性，除了在个别特别指明的地方之外。很自然，应该鼓励程序员去利用在大部分系统中可以使用的范围广泛的功能，但还是把它们留做练习。

3.2 Hello, world!

最小的C++程序是

```
int main() { }
```

它定义了一个称为`main`的函数，该函数没有参数，也不做任何事情。

每个C++程序中必须有一个名字为`main()`的函数，程序将从这个函数开始执行。由`main()`返回的`int`值，如果有的话，就是这个程序返回给“系统”的值。如果没有值被返回，系统将接到一个表示程序成功完成的值。来自`main()`的非0值表示出错。

典型情况是一个程序总要产生一些输出。这里是一个能够输出`Hello, World!`的程序：

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

行 `#include <iostream>` 指示编译器去包含位于 `iostream` 里的标准流 I/O 功能的声明。如果没有那些声明，表达式

```
std::cout << "Hello, world!\n"
```

将没有任何意义。运算符 `<<` (“放入”) 将它的第二个参数写到第一个参数里。在这一具体情况中，字符串文字量 `"Hello, world!\n"` 将被写进标准输出流 `std::cout`。字符串文字量就是由双引号括起来的一个字符序列。在字符串文字量中，由反斜线字符 `\` 后跟一个字符表示的是某个特殊字符。在这里出现的 `\n` 是换行字符，因此，串中的是字符 `Hello, world!`，后跟一个换行符。

3.3 标准库名字空间

标准库定义在一个称为 `std` 的名字空间里 (2.4 节、8.2 节)。这也就是为什么我写的是 `std::cout`，而不直接写 `cout` 的原因。我这样做是明确说出要使用的是标准 `cout`，而不是其他的什么 `cout`。

标准库的每种功能都是通过某个像 `<iostream>` 这样的头文件提供的。例如，

```
#include<string>
#include<list>
```

这就使标准的 `string` 和 `list` 都可以用了。借助于前缀 `std::` 就能使用它们：

```
std::string s = "Four legs Good; two legs Baaad!";
std::list<std::string> slogans;
```

为了简单起见，我将很少在实例中显式使用 `std::` 前缀，也将不总是去显式地 `#include` 必要的头文件。为了能编译和运行这里的程序片段，你必须自己去 `#include` 适当的头文件 (在 3.7.5 节、3.8.6 节和第 16 章列出)。除此之外，你还必须使用 `std::` 前缀，或者将出自 `std` 的每个名字都做成全局的 (8.2.3 节)。例如，

```
#include<string>                // 使标准字符串功能可以使用
using namespace std;           // 使所有std名字都可用，不必加std::前缀
string s = "Ignorance is bliss!"; // 可以：string就是std::string
```

一般来说，将一个名字空间中的所有名字统统倾倒进全局名字空间里，并不是一种好的做法。但是，为使这里用于阐释语言和库特征的实例比较短小，我还是忽略了那些将重复出现的 `#include` 和 `std::` 限定符。在这本书里，我几乎是只用到标准库。所以，如果用到了某个来自标准库的名字，那么，或者在那里用的就是标准库所提供的东西，或者就是为该项标准库功能可能如何定义提供一个解释。

3.4 输出

库 `iostream` 为每种内部类型定义了相应的输出方式。进一步说，为用户定义类型定义一种

输出方式也很容易。默认情况下，送到`cout`的输出值都将被转换为字符的序列。例如，

```
void f()
{
    cout << 10;
}
```

将把字符`1`而后是字符`0`放入标准输出流里。下面也一样：

```
void g()
{
    int i = 10;
    cout << i;
}
```

不同类型的输出可以按一种明显的方式组合在一起：

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

如果`i`的值是`10`，那么输出将是

`the value of i is 10`

字符常量的形式是单引号括起的一个字符。请注意，字符常量被输出为一个字符，而不是一个数值。例如，

```
void k()
{
    cout << 'a';
    cout << 'b';
    cout << 'c';
}
```

将输出`abc`。

人们很快就会对输出若干相关数据项时需要反复地写输出流的名字感到厌倦。幸好，一个输出表达式的结果本身还可以用于进一步的输出。例如，

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

这个函数与`h()`等价。关于流的更多细节将在第21章解释。

3.5 字符串

标准库提供了一个`string`类型，作为前面所用的字符串文字量的补充。这个`string`类型提供了许多很有用的字符串操作，例如串拼接等。请看这个例子：

```
string s1 = "Hello";
string s2 = "world";
```

```

void m1()
{
    string s3 = s1 + ", " + s2 + "!\n";
    cout << s3;
}

```

在这里，`s3`被初始化为如下的字符序列

Hello, world!

随后是一个换行符。对于字符串的加法表示的就是拼接。你可以将一个字符串，或一个字符串文字量，或一个字符加到一个字符串上。

在许多应用中，最常见的拼接形式是将某些东西追加到一个字符串的末尾。操作 `+=` 直接支持这种工作。例如

```

void m2(string& s1, string& s2)
{
    s1 = s1 + '\n'; // 追加换行符
    s2 += '\n';     // 追加换行符
}

```

这两种在字符串末尾追加的方式在语义上是等价的。但我喜欢后面一种，因为它更紧凑，实现效率也可能更高一些。

很自然，`string`可以相互比较，也可以与一个字符串文字量比较。例如，

```

string incantation;
void respond(const string& answer)
{
    if (answer == incantation) {
        // perform magic
    }
    else if (answer == "yes") {
        // ...
    }
    // ...
}

```

标准库的字符串类将在第20章中描述。在其他有用的特征中，还提供了操纵子字符串的功能。例如，

```

string name = "Niels Stroustrup";
void m3()
{
    string s = name.substr(6, 10); // s = "Stroustrup"
    name.replace(0, 5, "Nicholas"); // name变成 "Nicholas Stroustrup"
}

```

这里的`substr()`运算返回一个字符串，它就是函数参数所指明的那个子串的副本。第一个参数是在本字符串里的一个下标（位置），第二个参数是所需子串的长度。因为下标开始于0，所以`s`得到的值是`Stroustrup`。

`replace()`操作用一个值替换掉指定的子串。在上面情况中，开始于位置0长5的子串是`Niels`，它被用`Nicholas`取代。这样，`name`最后的值就是`Nicholas Stroustrup`。请注意，作为

替代的串不必与被它替换的子串一样长。

3.5.1 C风格的字符串

一个C风格的字符串就是一个以0字符结束的字符数组（5.5.2节）。如上所示，我们很容易将一个C风格的字符串放进一个`string`里。要调用以C风格字符串为参数的函数，我们就必须能够以C风格字符串的形式提取出`string`的值。函数`c_str()`能完成这件事（20.3.7节）。比如说，我们可以用C的输出函数`printf()`（21.8节）打印出`name`，方式如下：

```
void f()
{
    printf("name: %s\n", name.c_str());
}
```

3.6 输入

标准库为输入提供了`istream`。与`ostream`一样，`istream`能处理内部数据类型的字符序列表示。它也很容易扩充，以便去应付各种用户定义类型。

运算符`>>`（“取出”）被用做输入运算符；`cin`是标准输入流。`>>`右边运算对象的类型决定了可以接受什么输入，这个运算对象被作为输入操作的目标。例如，

```
void f()
{
    int i;
    cin >> i; // 把一个整数读到i

    double d;
    cin >> d; // 把一个双精度浮点数读到d
}
```

从标准输入读一个数，例如`1234`，放入整型变量`i`；再读一个浮点数，例如`12.34e5`，并将它放入双精度浮点变量`d`。

这里是一个例子，它执行从英寸到厘米以及从厘米到英寸的转换。你输入一个数，后面跟着一个表明单位（厘米或英寸）的字符，这个程序就会按照另一种单位输出对应的值。

```
int main()
{
    const float factor = 2.54; // 1英寸等于2.54厘米
    float x, in, cm;
    char ch = 0;

    cout << "enter length: ";

    cin >> x;        // 读入一个浮点数
    cin >> ch;        // 读入后缀

    switch (ch) {
    case 'i':        // 英寸
        in = x;
        cm = x*factor;
        break;
    case 'c':        // 厘米
        in = x/factor;
```

```

        cm = x;
        break;
    default:
        in = cm = 0;
        break;
    }

    cout << in << " in = " << cm << " cm\n";
}

```

这里的switch语句相对于一组常量去检查一个值。break语句用于跳出switch语句。各个分情况常量必须互不相同，*default*是可选的，程序员也不一定要提供*default*。

我们经常需要输入一系列字符。完成这件事的一个很方便的方式就是将它们读入一个*string*里。例如，

```

int main()
{
    string str;

    cout << "Please enter your name\n";
    cin >> str;
    cout << "Hello, " << str << "!\n";
}

```

如果你键入

Eric

回应将是

Hello, Eric!

按照默认方式，一个空白字符（5.2.2节），例如空格符，将结束一次输入。因此，如果你键入

Eric Bloodaxe

即使空格后的输入好像是York的那个超肥的国王，回答将仍然是

Hello, Eric!

你可以用函数*getline()*读入一个完整的行。例如，

```

int main()
{
    string str;

    cout << "Please enter your name\n";
    getline(cin, str);
    cout << "Hello, " << str << "!\n";
}

```

对于这个程序，输入

Eric Bloodaxe

就能将产生所预期的输出

Hello, Eric Bloodaxe!

这种标准字符串有着很好的性质，它可以自动扩展以存放你放进去的任意多的东西。所以，如果你送入了几百万个分号，这个程序也将送回你成页成页的分号——除非你的机器或者操

作系统在此之前用完了某种关键性的资源。

3.7 容器

许多计算都涉及到建立各种对象的汇集，以及对这些汇集的操作。逐个字符地读进一个字符串，逐个字符地将该串打印出来都是这种工作的实际例子。一个以保存一批对象为主要用途的类通常被称为一个容器。为给定的工作提供适当的容器，并提供一些有用的基本操作来支持它们，这些在任何程序的构造过程中都是最重要的步骤。

为阐释标准库中最有用的容器，现在让我们来考虑一个保存名字和电话号码的简单程序。它正是这样的一类程序，对于有不同背景的人而言，有许多不同的“简单而明显”的解决办法。

3.7.1 向量——*vector*

对于许多C程序员而言，(名字, 号码) 对的内部数组会被看做是一个合适的起点：

```
struct Entry {
    string name;
    int number;
};

Entry phone_book[1000];

void print_entry(int i)    // 简单使用
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

但无论如何，内部数组具有固定的规模。如果我们选择了某个很大的规模，那么就会浪费存储；而如果选择一个过小的规模，数组又会溢出。这两种情况都会使我们不得不去写低级的存储管理代码。标准库提供的*vector*（16.3节）能关照好所有这些情况

```
vector<Entry> phone_book(1000);

void print_entry(int i)    // 简单使用，就像数组一样
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}

void add_entries(int n) // 将其规模增加n
{
    phone_book.resize(phone_book.size()+n);
}
```

vector() 的成员函数*size*() 给出的是它的元素个数。

请注意在*phone_book*的定义中括号的使用。我们是做出了一个元素类型为*vector<Entry>*类型的对象，并通过一个初始值提出对它的规模要求。这与声明内部数组的方式很不一样：

```
vector<Entry> book(1000);    // 1000个元素的向量
vector<Entry> books[1000];  // 1000个空向量的数组
```

如果你真的犯了一个错误，在声明一个*vector*时将[]用到了你本来应该用()的地方，在你试图去使用这个*vector*时，几乎可以肯定你的编译器能够捕捉到这个错误，并发出一个错误信息。

vector是一种能赋值的简单对象。例如，

```
void f(vector<Entry>& v)
{
    vector<Entry> v2 = phone_book;
    v = v2;
    // ...
}
```

对**vector**赋值涉及到复制其中的所有元素。这样，在 $f()$ 初始化和赋值之后，在 v 和 $v2$ 里各保有**phone_book**中的各个**Entry**的一份副本。当向量里存有许多元素时，这种看起来无害的赋值和初始化的代价也可能很高。

3.7.2 范围检查

按照默认方式，标准库的**vector**并不提供对区间范围的检查（16.3.3节）。例如，

```
void f()
{
    int i = phone_book[100].number; // 100超出了区间范围
    // ...
}
```

这个初始化很可能将某个随机的值赋给 i ，而不是给出一个错误。这当然不是人们所希望的，所以我在下面各章里要使用**vector**的一个经过简单修订的版本，它带有区间范围检查，称为**Vec**。一个**Vec**就像是一个**vector**，只是当某个下标超出了区间范围时，它将抛出一个类型为**out_of_range**的异常。

实现像**Vec**这样的类型以及有效地使用异常的技术将在11.12节、8.3节和第14章讨论。但无论如何，下面的定义对于本书中的所有例子都足够：

```
template<class T> class Vec : public vector<T> {
public:
    Vec() : vector<T>() {}
    Vec(int s) : vector<T>(s) {}

    T& operator[] (int i) { return at(i); } // 检查区间范围
    const T& operator[] (int i) const { return at(i); } // 检查区间范围
};
```

这里的 $at()$ 操作是**vector**的下标操作，如果它的参数超出了该**vector**的区间范围， $at()$ 就会抛出一个**out_of_range**类型的异常（16.3.3节）。

现在回到保存名字和电话号码的问题。我们现在用一个**Vec**来保证所有超出范围的访问都将被抓住。例如，

```
Vec<Entry> phone_book(1000);
void print_entry(int i) // 简单地使用，完全像用Vector
{
    cout << phone_book[i].name << ' ' << phone_book[i].number << '\n';
}
```

一个超范围的访问将抛出一个异常，用户可以捕捉到它。例如，

```
void f()
{
```

```

    try {
        for (int i = 0; i < 10000; i++) print_entry(i);
    }
    catch (out_of_range) {
        cout << "range error\n";
    }
}

```

在企图用*i* = 1000访问`phone_book[i]`时，异常将被抛出，而后被捕捉。

如果用户未捕捉这类异常，程序就将按照一种定义良好的方式终止，而不会继续下去，也不会以某种未定义的方式失败。使异常所造成的奇怪现象最小化的一种方式是让`main()`采用一个`try`-块作为体：

```

int main()
try {
    // 你的代码
}
catch (out_of_range) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}

```

在这里提供了一个默认的异常处理器。这样，如果我们未能捕捉某些异常，它就会在标准的错误诊断输出流`cerr`（21.2.1节）打印出一个错误信息。

3.7.3 表——*list*

对电话簿的插入和删除有可能很频繁。这样，对于表示一个简单的电话簿而言，采用表就可能比采用向量更为合适。例如写：

```
list<Entry> phone_book;
```

如果使用的是表，我们将倾向于不采用通过下标访问元素，像对向量的一般做法那样，而可能是检索这个表，去寻找具有某个给定值的元素。在这样做时，我们利用了这样的事实：表也是一个序列，如3.8节所述。

```

void print_entry(const string& s)
{
    typedef list<Entry>::const_iterator LI;
    for (LI i = phone_book.begin(); i != phone_book.end(); ++i) {
        const Entry& e = *i; // 采用引用是为了方便
        if (s == e.name) {
            cout << e.name << ' ' << e.number << '\n';
            return;
        }
    }
}

```

对*s*的检索从表的起始处开始，一直进行到*s*被找到，或是达到了表的结束位置。每个标准库容器都提供了函数`begin()`和`end()`，它们分别返回到容器中第一个元素和超过末尾一个元素的迭代器（16.3.2节）。对于一个给定的迭代器*i*，相应的下一个元素是`++i`，它所引用的元素是`*i`。

用户不必知道标准容器的迭代器的确切类型。这种迭代器的类型是容器定义的一部分，

可以通过名字引用。当我们不需要修改容器的元素时，*const_iterator*就是我们所需要的类型。如果要修改元素，我们就应该用普通的*iterator*类型（16.3.1节）。

向一个*list*加入一个元素也很容易：

```
void add_entry(Entry& e, list<Entry>::iterator i)
{
    phone_book.push_front(e);    // 加在开头
    phone_book.push_back(e);     // 加在最后
    phone_book.insert(i, e);     // 加在i所引用的元素之前
}
```

3.7.4 映射——*map*

写出一些代码，在一个（名字，号码）对的表中寻找一个名字，这也是一件很讨厌的事情。除此之外，线性检索一般也是相当低效的，除非对于特别短的表。另有一些数据结构直接支持插入、删除和基于值的检索，标准库特别为此提供了*map*类型（17.4.1节）。*map*就是值的对偶的容器。例如，

```
map<string, int> phone_book;
```

在其他一些地方，人们也把*map*称做关联数组或者字典。

在用它的一个类型（称为关键码）的某个值去索引时，*map*将返回其第二个类型的（称为值类型，或者映射类型）的对应值。例如，

```
void print_entry(const string& s)
{
    if (int i = phone_book[s]) cout << s << " " << i << "\n";
}
```

如果对于某个关键码*s*无法找到匹配，*phone_book*将返回一个默认值。*map*里对整数类型的默认值是0。这里我实际上假定了0不是一个合法的电话号码。

3.7.5 标准容器

map、*list*或*vector*都可以用于表示一个电话簿。然而，它们中的每一个都有其长处和短处。例如，对向量的下标操作开销较小且易于操作，而另一方面，在向量的两个元素之间插入一个元素则是代价高昂的。*list*恰好具有与此相反的性质。*map*类似于（关键码，值）对偶的表，除此之外它还针对基于关键码去查询值做了特殊的优化。

标准库提供了一些最普通最常用的容器类型，这就使程序员可以选择某种能最好地满足其实际应用需要的容器：

标准库容器的总结	
<i>vector</i> < <i>T</i> >	变长向量（16.3节）
<i>list</i> < <i>T</i> >	双向链表（17.2.2节）
<i>queue</i> < <i>T</i> >	队列（17.3.2节）
<i>stack</i> < <i>T</i> >	堆栈（17.3.1节）
<i>deque</i> < <i>T</i> >	双端队列（17.2.3节）
<i>priority_queue</i> < <i>T</i> >	按值排序的队列（17.3.3节）
<i>set</i> < <i>T</i> >	集合（17.4.3节）
<i>multiset</i> < <i>T</i> >	集合，值可重复出现（17.4.4节）
<i>map</i> < <i>key</i> , <i>value</i> >	关联数组（17.4.1节）
<i>multimap</i> < <i>key</i> , <i>value</i> >	关联数组，关键字可重复出现（17.4.2节）

标准容器将在16.2节、16.3节和第17章中讨论。这些容器都定义在名字空间`std`里，在`<vector>`、`<list>`、`<map>`等头文件里面描述（16.2节）。

从记法的角度看，标准容器及其基本操作都被设计成相互类似的。进一步地，在不同容器里同样操作的意义也是等价的。一般来说，许多基本操作适用于各种种类的容器。举例来说，`push_back()`能用于（以合理的效率）将一个元素加到一个`vector`或一个`list`的最后，每个容器都有一个`size()`成员函数，它们都返回容器中元素的个数。

这种记法和语义的一致性也使程序员可以提供新的容器类型，并使它们能按照与标准容器类似的方式使用。检查范围的向量`Vec`（3.7.2节）也就是这样的例子。第17章将展示如何把`hash_map`加入这个框架之中。容器界面的一致性还使我们能描述各种不依赖于任何特定容器类型的算法。

3.8 算法

一个数据结构，例如表或向量，仅就自身而言也没有太大用处。为了使用它们，我们就需要有最基本的访问操作，例如加入或者删除元素等。进一步说，我们也很少只是将元素存储进去，而是需要对它们排序，打印它们，抽取某些子集，删除一些元素，检索特定的对象，如此等等。正因为此，标准库除提供了各种最常用的容器类型之外，还提供了一批用于这些容器的最常用的算法。例如，下面程序对一个`vector`排序，并将该`vector`中元素复制到一个`list`，其中的重复元素只复制唯一的副本：

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), le.begin());
}
```

标准算法将在第18章描述，它们都是基于元素的序列（2.7.2节）表述的。一个序列由一对迭代器表示，一个描述其首元素，另一个描述超出序列末端一个位置的元素。在上面这个例子里，`sort()`对由`ve.begin()`到`ve.end()`的序列做排序，这也正好就是该`vector`的所有元素。为了向序列里写，你只要描述需要写的第一个元素。如果要写进去的元素不止一个，那么跟随在这个初始元素之后的一些元素也将被复写。

如果我们想在一个容器的最后增加一个元素，那就应该写：

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    sort(ve.begin(), ve.end());
    unique_copy(ve.begin(), ve.end(), back_inserter(le)); // 附到le之后
}
```

`back_inserter()`在容器的最后增加元素，并根据需要扩充这个容器，以便为新增加的元素提供空间（19.2.4节）。这样，标准容器和`back_inserter()`的结合使我们能避免去使用极易出错的，采用`realloc()`（16.3.5节）的C风格存储管理。如果要在后面附加元素，但却忘记用`back_inserter()`就会导致错误。例如，

```
void f(vector<Entry>& ve, list<Entry>& le)
{
    copy(ve.begin(), ve.end(), le);           // 错误：le不是迭代器
    copy(ve.begin(), ve.end(), le.end());      // 糟糕：写超出结束位置
```

```
    copy(v.begin(), v.end(), le.begin()); // 覆盖掉一些元素
}
```

3.8.1 迭代器的使用

当你第一次遇到某个迭代器时，总能得到几个引用着最有用的元素的迭代器：*begin()* 和 *end()* 是其中最好的例子。除此之外，许多算法也将返回迭代器。例如，标准算法 *find* 在一个序列里查找某个值，并返回引用着找到的那个元素的迭代器。利用 *find*，我们可以统计出某一字符在一个 *string* 中出现的次数：

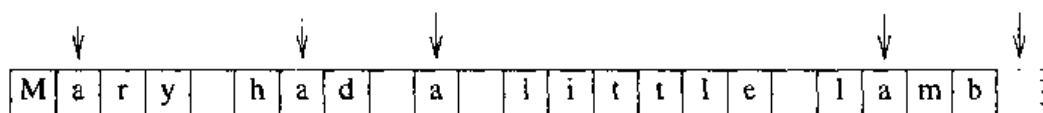
```
int count(const string& s, char c) // 统计出c在s里出现的次数
{
    int n = 0;
    string::const_iterator i = find(s.begin(), s.end(), c);
    while (i != s.end()) {
        ++n;
        i = find(i+1, s.end(), c);
    }
    return n;
}
```

算法 *find* 返回一个引用着有关值在序列里的第一次出现的迭代器，或者是超过结束处一个位置的迭代器。现在考虑对 *count* 的简单调用中会出现什么情况：

```
void f()
{
    string m = "Mary had a little lamb";
    int a_count = count(m, 'a');
}
```

对 *find()* 的第一次调用将发现 *Mary* 里的 'a'。这时返回的迭代器将指向这个字符，因而不是 *s.end()*，所以我们就进入了循环。在循环中，我们又从 *i + 1* 开始检索；也就是说，从所发现的 'a' 之后一个位置开始。随后的循环发现了另外三个 'a'。完成了所有这些之后，*find()* 到达了结束处并返回 *s.end()*，这使条件 *i != s.end()* 为假，并使循环退出。

这里对 *count()* 的调用可以用图形方式表示如下



这里的箭头表示的是迭代器 *i* 的初始的、中间的和最后的位置。

很自然，算法 *find* 对于每个标准容器的工作都完全相同。根据这个情况，我们也可以按照同样的方式推广 *count()* 函数：

```
template<class C, class T> int count(const C& v, T val)
{
    typename C::const_iterator i = find(v.begin(), v.end(), val); // "typename" 参见C.13.5节
    int n = 0;
    while (i != v.end()) {
        ++n;
        ++i; // 跳过刚刚发现的元素
        i = find(i, v.end(), val);
    }
}
```



```

    }
    return n;
}

```

这样做能行，因此我们可以说

```

void f(list<complex>& lc, vector<string>& vs, string s)
{
    int i1 = count(lc, complex(1, 3));
    int i2 = count(vs, "Diogenes");
    int i3 = count(s, 'x');
}

```

但是我们并不需要去定义`count`模板。由于统计出元素的出现次数是如此具有一般性的有用功能，标准库提供了这个算法。为了达到完全的通用性，标准库的`count`以一个序列作为参数，而不是以容器作为参数，因此我们应该采用如下的写法：

```

void f(list<complex>& lc, vector<string>& vs, string s)
{
    int i1 = count(lc.begin(), lc.end(), complex(1, 3));
    int i2 = count(vs.begin(), vs.end(), "Diogenes");
    int i3 = count(s.begin(), s.end(), 'x');
}

```

采用序列的写法，将使我们也能把`count`用于内部数组，还可以对容器中的一部分做统计。例如，

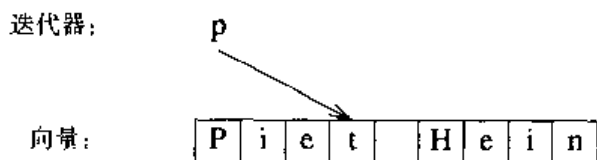
```

void g(char cs[], int sz)
{
    int i1 = count(&cs[0], &cs[sz], 'z'); // 数组中的 'z'
    int i2 = count(&cs[0], &cs[sz/2], 'z'); // 数组前半一半中的 'z'
}

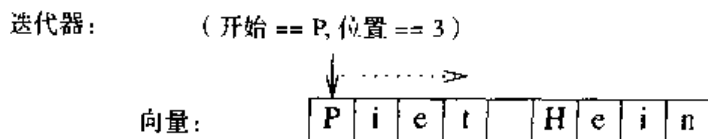
```

3.8.2 迭代器类型

迭代器实际上是什么呢？任何特定的迭代器也都是某个类型的对象。当然，存在着许许多多不同的迭代器类型，因为每个迭代器都需要保存为在一个特定容器上完成自己的工作所需要的信息。这些迭代器的类型互相之间的差异可以像容器一样巨大，而且都是为某项特定工作专门度身打造的。例如，`vector`的迭代器或许就是一种常规的指针，因为指针是引用`vector`中元素的一种很合理的方式：

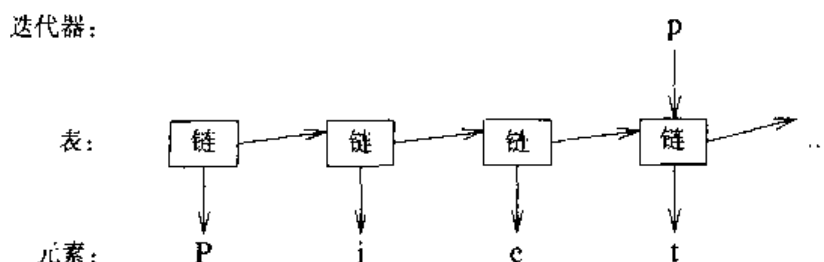


换种方式，`vector`的迭代器也可以被实现为一个到`vector`的指针，再加一个下标：



采用这种迭代器就能实现对区间范围的检查。

一个表迭代器必须是某种比指向元素的简单指针更复杂些的东西，因为一般来说，表中的一个元素根本不知道下一个元素在哪里。这样，表迭代器可以是一个指向链接的指针：



所有迭代器的共同之处在于它们的语义，以及它们的有关操作的名字。例如，对一个迭代器使用 `++` 操作总产生引用下一个元素的迭代器。与此类似，`*` 总产生被这个迭代器所引用的那个元素。事实上，任何对象只要能遵守这样为数不多的几条原则，它就是一个迭代器了（19.2.1节）。进一步说，用户极少需要知道一个特定迭代器的具体类型；每个容器都“知道”它的迭代器的类型，并采用约定好了名字 `iterator` 和 `const_iterator`，使它们可以在其他地方使用。例如，`list<Entry>::iterator` 就是 `list<Entry>` 中普通迭代器的类型。我极少需要去关心这些类型是如何定义的。

3.8.3 迭代器和I/O

迭代器是一种处理位于容器中的元素序列的非常一般而又极其有用的概念。但是，容器并不是我们能够遇到元素序列的仅有的地方。例如，一个输入流也能产生一个值的序列，我们也会把一个值的序列写进一个输出流中。正因为这样，人们也将迭代器的概念很有效地应用到输入和输出上。

要做出一个 `ostream_iterator`，我们需要描述被使用的将是哪个流，还要描述写入其中的对象的类型。例如，我们可以定义一个引用了标准输出流 `cout` 的迭代器：

```
ostream_iterator<string> oo(cout);
```

给 `*oo` 赋值的效果就是使被赋的值送到 `cout`。例如，

```
int main()
{
    *oo = "Hello, ";    // 意思是 cout << "Hello, "
    ++oo;
    *oo = "world!\n";   // 意思是 cout << "world!\n"
}
```

这就形成另一种向标准输出写规范信息的方式。这里 `++oo` 的记法是模仿通过指针向数组写入的方式。对于简单的工作，这种方式不会是我的第一选择。但把输出处理为一种只能写入的容器是很有用的，我们不久就会看得更清楚——如果现在还没完全明白的话。

与此类似，一个 `istream_iterator` 就是某种东西，它使我们像从容器读出一样从输入流中读出。同样，我们必须描述所用的输入流和所期望的值类型：

```
istream_iterator<string> ii(cin);
```

由于输入迭代器总是成对出现，以表示一个序列，因此我们也就必须提供另一个 `istream_iterator`

去表示输入的结束。这就是那个默认的`istream_iterator`:

```
istream_iterator<string> eos;
```

现在我们就从输入流读入`Hello, world!`，而后再次将它写出去:

```
int main()
{
    string s1 = *ii;
    ++ii;
    string s2 = *ii;

    cout << s1 << ' ' << s2 << '\n';
}
```

实际中，`istream_iterator`和`ostream_iterator`并不是想供人们直接使用的，它们主要是为了给算法提供参数。例如，我们可以写出一个简单程序，它读入一个文件，对所读的东西排序，去掉重复，最后将结果写入另一个文件:

```
int main()
{
    string from, to;
    cin >> from >> to;                // 取得源文件名和目标文件名

    ifstream is(from.c_str());          // 输入流 (c_str(); 见3.5.1节和20.3.7节)
    istream_iterator<string> ii(is);     // 流的输入迭代器
    istream_iterator<string> eos;        // 输入的哨兵

    vector<string> b(ii, eos);           // b是一个向量，用输入进行初始化
    sort(b.begin(), b.end());           // 对缓冲区排序

    ofstream os(to.c_str());            // 输出流
    ostream_iterator<string> oo(os, "\n"); // 流的输出迭代器

    unique_copy(b.begin(), b.end(), oo); // 从缓冲区复制到输出
                                         // 并去掉重复的值

    return !is.eof() || !os;            // 返回错误信息 (3.2节, 21.3.3节)
}
```

`ifstream`就是可以附着到文件上的`istream`，`ofstream`是可以附着到文件上的`ostream`。`ostream_iterator`的第二个参数用于分隔各个输出值。

3.8.4 遍历和谓词

迭代器使我们能写出迭代穿过一个序列的循环。当然，总写循环也会使人厌倦，为此，标准库提供了一些能对序列中的每个元素调用一个函数的方法。

考虑去写一个程序，它从输入中读一些单词，并记录下它们出现的频率。表示字符串及其相关频率的最明显方式是采用一个`map`

```
map<string, int> histogram;
```

对每个串记录其频率，需要做的明显操作是

```
void record(const string& s)
{
    histogram[s]++;    // 记录"s"出现的次数
}
```

一旦读完了输入，我们还可能想要输出工作中收集到的数据。这个`map`由一个 `(string, int)` 对的序列组成，因此，我们就希望对`map`中的每个元素调用下面函数：

```
void print(const pair<const string, int>& r)
{
    cout << r.first << ' ' << r.second << '\n';
}
```

(`pair`的第一个元素被称为`first`，第二个元素被称为`second`)。`pair`的第一个元素是个`const string`，而不是普通的`string`，因为`map`里所有的关键码都是常量。

这样，主程序就可以写成：

```
int main()
{
    istream_iterator<string> ii(cin);
    istream_iterator<string> eos;

    for_each(ii, eos, record);
    for_each(histogram.begin(), histogram.end(), print);
}
```

请注意，我们不需要对`map`排序以便使输出是按顺序的，因为`map`总是按顺序保存它的元素，因此，它的迭代器也将按（上升）序遍历`map`的所有元素。

许多程序设计工作都涉及到在容器中寻找某些东西，而不是简单地对每个元素做某件事情。例如，`find`算法（18.5.2节）是为寻找某个特定值提供的一种很方便的方式。这一思想的另一种更一般的变形是寻找满足某种特殊需要的元素。例如，我们可能需要在在一个`map`里寻找第一个大于42的值。一个`map`就是一个（关键码，值）对偶的序列，因此我们需要检索这个序列，去找一个`pair<const string, int>`，其中的`int`大于42：

```
bool gt_42(const pair<const string, int>& r)
{
    return r.second > 42;
}

void f(map<string, int>& m)
{
    typedef map<string, int>::const_iterator MI;
    MI i = find_if(m.begin(), m.end(), gt_42);
    // ...
}
```

换一种情况，我们也可能需要统计频率高于42的单词的个数：

```
void g(const map<string, int>& m)
{
    int c42 = count_if(m.begin(), m.end(), gt_42);
    // ...
}
```

像`gt_42()`这样用于控制算法的函数被称为谓词。谓词将被针对每个元素调用并返回布尔值，算法根据这种返回值去执行自己预定的动作。例如，`find_if()`将检索到它的谓词返回`true`为止，这说明找到了一个要找的元素。与此类似，`count_if()`统计出它的谓词返回`true`的次数。

标准库提供了若干很有用的谓词，以及一些能用于构造出更多谓词的模板（18.4.2节）。

3.8.5 使用成员函数的算法

有许多算法都是将某个函数应用于一个序列里的各个元素。例如，在3.8.4节

```
for_each(ii, eos, record);
```

对输入的对每个串调用*record*()。

我们也经常需要处理指针的容器，而且实际希望的是对每个被指向的元素调用它的某个成员函数，而不是对指针去使用一个全局函数。例如，我们可能想对一个*list<Shape*>* 里的每个元素调用成员函数*Shape::draw*()。要处理这个特定例子，我们先简单写出一个非成员函数，让它去调用成员函数。例如，

```
void draw(Shape* p)
{
    p->draw();
}

void f(list<Shape*>& sh)
{
    for_each(sh.begin(), sh.end(), draw);
}
```

通过将这种技术加以推广，我们就可以按如下方式写出这个例子：

```
void g(list<Shape*>& sh)
{
    for_each(sh.begin(), sh.end(), mem_fun(&Shape::draw));
}
```

标准库模板*mem_fun*() (18.4.4.2节) 以一个到成员函数的指针 (15.5节) 为参数，产生出某种东西，使它可以对某个指向该成员所在的类的指针调用。这里*mem_fun*(&*Shape::draw*) 的结果就以*Shape** 为参数，返回的就是*Shape::draw*() 返回的东西。

这种*mem_fun*() 机制是非常重要的，因为它使各种标准算法能够被应用于保存多态对象的容器。

3.8.6 标准库算法

什么是一个算法？算法的一个一般性的定义是“一组有穷的规则，它给出了为解决一个特定问题集合的一个操作序列，[并]包含五个重要特征：有穷性……定义性……输入……输出……效率” [Knuth, 1968, 1.1节]。在C++标准库的环境里，一个算法就是一组在元素序列上操作的模板。

标准库提供了数十个算法，这些算法都定义在名字空间*std*里，在头文件<*algorithm*>里描述。下面是其中几个我觉得特别有用的东西：

标准库算法的选择	
<i>for_each</i> ()	对每个元素调用函数 (18.5.1节)
<i>find</i> ()	找出参数的第一个出现 (18.5.2节)
<i>find_if</i> ()	找出第一个满足谓词的元素 (18.5.2节)
<i>count</i> ()	统计元素的出现次数 (18.5.3节)
<i>count_if</i> ()	统计与谓词匹配的元素 (18.5.3节)
<i>replace</i> ()	用新值取代元素 (18.6.4节)
<i>replace_if</i> ()	用新值取代满足谓词的元素 (18.6.4节)
<i>copy</i> ()	复制元素 (18.6.1节)
<i>unique_copy</i> ()	复制元素，不重复 (18.6.1节)
<i>sort</i> ()	对元素排序 (18.7.1节)
<i>equal_range</i> ()	找到所有具有等价值的元素 (18.7.2节)
<i>merge</i> ()	归并排序的序列 (18.7.3节)

这些算法以及其他许多算法（见第18章）都可以用于容器、*string*和内部数组的元素。

3.9 数学

像C一样，在C++的基本设计中并没有把数值计算放在心上。然而，确有许多数值工作是在C++中做的，标准库也反映了这种情况。

3.9.1 复数

标准库支持一族复数类型，采用的也是类似于在2.5.2节所描述*complex*类的方式。为了支持其标量可以是单精度浮点数（*float*），双精度数（*double*）等的复数，标准库提供的*complex*是一个模板：

```
template<class scalar> class complex {
public:
    complex(scalar re, scalar im);
    // ...
};
```

它支持复数类型的常规算术运算和最常用的数学函数。例如，

```
// 取自<complex>的标准指数函数：
template<class C> complex<C> pow(const complex<C>&, int);
void f(complex<float> fl, complex<double> db)
{
    complex<long double> ld = fl+sqrt(db);
    db += fl*3;
    fl = pow(1/fl, 2);
    // ...
}
```

更多的细节参见22.5节。

3.9.2 向量算术

在3.7.1节里描述的*vector*的设计是为了作为一种保存值的一般性机制，那里追求的是灵活性，以及能融入容器、迭代器和算法的体系结构之中。但是它并不支持数学的向量运算。将这些运算添加到*vector*上并不难，但是向量的一般性和灵活性会妨碍优化，而优化对于所有重要的数值工作都是最基本的东西。由于这一情况，标准库提供了一种称为*valarray*的向量。它不那么通用，但却更容易适应数值计算的优化

```
template<class T> class valarray {
    // ...
    T& operator[] (size_t);
    // ...
};
```

类型*size_t*是实现中用于表示数组下标的无符号整数类型。

这里也支持*valarray*的常规算术运算和最常用的数学函数。例如，

```
// 取自<valarray>的标准的绝对值函数
template<class T> valarray<T> abs(const valarray<T>&);
```

```

void f(valarray<double>& a1, valarray<double>& a2)
{
    valarray<double> a = a1*3.14+a2/a1;
    a2 += a1*3.14;
    a = abs(a);
    double d = a2[7];
    // ...
}

```

更多的细节见22.4节。

3.9.3 基本数值支持

很自然，标准库中还包括对浮点数的最常用的数学函数——例如，`log()`，`pow()`和`cos()`等；见22.3节。除此之外，这里还提供了描述内部类型性质的类——例如，`float`的最大指数是多少；见22.2节。

3.10 标准库功能

标准库所提供的功能可以分类如下：

- [1] 基本运行支持（例如，存储分配和运行时类型信息等）；见16.1.3节。
- [2] C标准库（做了极少的修改，以便尽可能地减少其中违反类型系统的情况）；见16.1.2节。
- [3] 字符串和I/O流（包括国际化字符集和本地化）；见第20、21章。
- [4] 容器（例如`vector`、`list`和`map`）和使用容器的算法（例如一般性的遍历、排序和归并等）的框架；见第16、17、18和19章。
- [5] 对数值计算的支持（带算术运算的复数和向量，类似BLAS的和推广的切割，为容易进行优化而设计的语义）；见第22章。

将一个类包括进标准库的主要准则是它很可能会被每个C++程序员使用（无论是专家，还是初学者）；它能够以一种具有普遍性的形式提供，而且与同样功能的简单版本相比又不会增加明显的额外开销；它的简单使用是比较容易学习的。从本质上说，C++标准库提供了最常见的基本数据结构，以及在它们上面操作的基本算法。

每种算法都可以对任何容器使用，而无须做任何转换。这个框架一般被称为STL [Stepanov, 1994]，它也是可扩展的。这里所说的扩展是指，用户可以很容易地提供各种容器和算法，可以将它们添加到作为标准库的一部分所提供那些东西之中，并使它们能直接与标准库容器和算法一起工作。

3.11 忠告

- [1] 不要像重新发明车轮那样企图做每件事；去使用库。
- [2] 不要相信奇迹；要理解你的库能做什么，它们如何做，它们做时需要多大的代价。
- [3] 当你遇到一个选择时，应该优先选择标准库而不是其他的库。
- [4] 不要认为标准库对于任何事情都是最理想的。
- [5] 切记 `#include` 你所用到的功能的头文件；3.3节。
- [6] 记住，标准库的功能定义在名字空间 `std` 之中；3.3节。

- [7] 请用`string`，而不是`char*`；3.5节、3.6节。
- [8] 如果怀疑，就用一个检查区间范围的向量（例如`Vec`）；3.7.2节。
- [9] `vector<T>`、`list<T>` 和 `map<key, value>` 都比 `T[]` 好；3.7.1节、3.7.3节、3.7.4节。
- [10] 如要向一个容器中添加一个元素，用`push_back()`或`back_inserter()`；3.7.3节、3.8节。
- [11] 采用对`vector`的`push_back()`，而不是对数组的`realloc()`；3.8节。
- [12] 在`main()`中捕捉公共的异常；3.7.2节。

第一部分 基本功能

这一部分描述C++ 的内部类型以及由它们出发构造程序的基本功能。在这里介绍的是C++ 的C子集，再加上C++ 对传统程序设计风格的进一步支持，还要讨论为通过若干逻辑的和物理的部分组合产生C++ 程序而提供的一些基本功能。

章目

- 第4章 类型和声明
- 第5章 指针、数组和结构
- 第6章 表达式和语句
- 第7章 函数
- 第8章 名字空间和异常
- 第9章 源文件和程序

第4章 类型和声明

绝不要接受任何不完美之物！

——无名氏

只有到崩溃的那一刻
才可能达到完美。

——C. N. Parkinson

类型——基本类型——布尔量——字符——字符文字量——整数——整数文字量——浮点类型——浮点文字量——大小——*void*——枚举——声明——作用域——初始化——对象——*typedef*——忠告——练习

4.1 类型

考虑

```
x = y + f(2);
```

要使这种东西在C++ 程序里面有意义，名字 x 、 y 和 f 就需要有合适的声明。这也就是说，程序员必须描述分别命名为 x 、 y 和 f 实体的存在性，而且它们的类型能使 $=$ （赋值）、 $+$ （加）以及 $()$ （函数调用）分别有意义。

在C++ 程序里，每个名字都有一个与之相关联的类型，这个类型决定了可以对这个名字应用什么操作（即应用于这个名字所指称的实体），并决定这些操作将如何做出解释。例如，声明

```
float x;           // x是浮点变量
int y = 7;         // y是整型标量，且有初始值7
float f(int);      // f是函数，有一个int型的参数，返回浮点数值
```

将能使上面的那个例子有意义。因为 y 被声明为 int ，所以它可以被赋值，可以用在算术表达式里，如此等等。另一方面， f 被声明为一个以 int 为参数的函数，所以可以用合适的参数去调用它。

本章将介绍各种基本类型（4.1.1节）和声明（4.9节）。这里的例子只是为了阐释各种语言特征，并不想做什么有意义的事情。范围更广阔且更实际的例子留到后面各章，留到描述了C++更多的东西之后。本章只提供最基本的元素，C++ 程序将从这些东西中构造起来。你必须知道这些东西，再加上伴随它们的术语和简单的语法，以便能够在C++ 里完成真正的项目，能够去阅读其他人写的代码。然而，要理解后面各章，并不需要彻底理解本章中所提到的每一个细节。因此，你可以按自己的喜好略读本章，只留意其中最主要的概念，等到以

后根据需要再转回来理解进一步的细节。

4.1.1 基本类型

C++有一组基本类型，它们对应于计算机的基本存储单元和使用这些单元去保存数据的一些最常见方式：

4.2节 一个布尔类型 (*bool*)。

4.3节 字符类型 (例如*char*)。

4.4节 整数类型 (例如*int*)。

4.5节 浮点类型 (例如*double*)。

除此之外，用户还可以定义

4.8节 表示一组特定值的枚举类型 (*enum*)。

这里还有

4.7节 类型*void*，用于表示没有信息。

从这些类型出发，我们可以构造出其他类型：

5.1节 指针类型 (例如*int**)。

5.2节 数组类型 (例如*char[]*)。

5.5节 引用类型 (例如*double&*)。

5.7节 数据结构和类 (第10章)。

我们将布尔量、字符和整数类型放到一起称为整型 (*integral type*)。整型和浮点类型一起称为算术类型。枚举和类 (第10章) 被称为用户定义类型，因为它们必须由用户定义出来，而不能事先没有声明就直接使用，而这正是那些基本类型的情况。与用户定义类型相对应，其他类型都被称为内部类型。

整型和浮点类型都提供了多种不同的尺寸，以便在所占用的存储量、表示精度和计算的可能范围等诸方面，给程序员提供一个选择的机会 (4.6节)。这里的假设是，计算机提供了字节以存放字符，提供了机器字以存放并计算整数值，提供了某些最适合存放浮点数的实体，还有地址，通过它可以引用所有这些实体。C++的基本类型，再加上指针和数组，以一种与具体机器无关的方式，为程序员呈现了这些位于机器层面上的概念。

对大部分应用而言，你可以简单地用*bool*表示逻辑值，用*char*表示字符，用*int*表示整数，用*double*表示浮点值。其他基本类型都是为优化或特殊需要而提供的变化，在真正需要它们之前最好是忽略之。当然，必须知道它们，以便能够阅读已有的C和C++代码。

4.2 布尔量

一个布尔量，*bool*，可以具有两个值*true*或*false*之一。布尔量用于表示逻辑运算的结果。例如，

```
void f(int a, int b)
{
    bool b1 = a==b;    // =是赋值，==是相等判断
    // ...
}
```

如果*a*和*b*具有相同的值，*b1*将变成*true*；否则*b1*将变成*false*。

bool最常见的使用是作为检查某些条件是否成立的函数（谓词）的结果类型。例如，

```
bool is_open(File*);
bool greater(int a, int b) { return a>b; }
```

按照定义，**true**具有值1，而**false**具有值0。与此相对应，整数可以隐式地转换到**bool**值：非零的整数转换为**true**，而0转换为**false**。例如，

```
bool b = 7;      // bool(7) 是true，所以b变成true
int i = true;    // int(true) 是1，所以i变成1
```

在算术和逻辑表达式里，**bool**都将被转为**int**，在这种转换之后得到的值上进行各种算术和逻辑运算。如果结果又被转回**bool**，那么0将转为**false**，所有非零值都转为**true**。

```
void g()
{
    bool a = true;
    bool b = true;

    bool x = a+b;  // a+b是2，所以x变成true
    bool y = a|b;  // a|b是1，所以y变成true
}
```

指针也可以隐式地转换到**bool**（C.6.2.5节），非零指针转为**true**，具有零值的指针转为**false**。

4.3 字符类型

类型为**char**的变量可以保存具体实现所用的字符集里的一个字符。例如，

```
char ch = 'a';
```

实际情况中一个**char**类型几乎都包含8个二进制位，因此它可以保存256种不同的值。典型的情况是有关字符集采用ISO-646的某个变形，例如ASCII，并由此提供了你键盘上的那些字符。这一字符集只是部分地标准化了，由此也引起了许多问题（C.3节）。

在支持不同自然语言的字符集之间存在着巨大的差异，以不同方式支持同一种自然语言的字符集之间也有类似情况。当然，我们在这里只对这些差异如何影响C++感兴趣，怎样在多语言、多字符集的环境中编写程序的事情已经超出了本书的范围，虽然这种事情也会在一些地方向我们招手（20.2节、21.7节、C.3.3节）。

假定实现所用的字符集包括数字、26个英文字母，以及某些基本标点符号是没问题的。而做出如下假定都是不安全的：在8位字符集中共有不超过127个字符（例如，有的字符集提供了255个字符），不存在超出英语的字母（大部分欧洲语言都提供了更多的字母），字母字符是连续排列的（EBCDIC在'i'和'j'之间留有空隙），写C++所需要的每个字符都是可用的（例如，有些国家的字符集中没有提供{ } [] \；C.3.1节）。只要有可能，我们都应该避免做出有关对象表示方式的任何假定，这个普遍规则甚至也适用于字符。

每个字符常量有一个整数值。例如，在ASCII字符集里'b'的值是98。这里是一个小程序，它可以告诉你，你仔细输入的任一个字符所对应的整数值是什么

```
#include <iostream>

int main()
{
    char c;
```

```

std::cin >> c;
std::cout << "the value of ' " << c << " ' is " << int(c) << '\n';
}

```

记法`int(c)`将给出字符`c`的整数值。能够把`char`转为整数也引起了一个问题：一个`char`是有符号的还是没符号的？由8个二进制位表示的256个值可以解释为整数值0~255，或者解释为-128~127。不幸的是，关于普通`char`如何选择的问题是由实现决定的（C.1节、C.3.4节）。C++提供了另外两个类型，它们都确切地回答了这个问题：*signed char*保存的值是-128~127；而*unsigned char*保存的值是0~255。幸运的是，这方面的差异只出现在那些超出127的值，而最常用的字符都在127之内。

将超过上述范围的值存入普通的`char`将会导致微妙的移植性问题。如果你真的需要使用不止一种`char`类型，或者你需要将整数存储到`char`中，那么请读一读C.3.4节。

这里还提供了另一个类型`wchar_t`，用于保存更大的字符集里的字符，例如Unicode的字符。这是另外一个独立的类型，`wchar_t`的大小由实现确定，且保证足够存放具体实现所用的现场（21.7节、C.3.3节）所支持的最大的字符集。这个奇怪的名字来源于C。在C语言里，`wchar_t`是一个typedef（4.9.7节）而不是一个内部类型。加上后缀`_t`就是为了区分标准类型和typedef。

请注意，字符类型都是整型（4.1.1节），可以对它们使用算术和逻辑运算符（6.2节）。

4.3.1 字符文字量

字符文字量常被称做字符常量，其形式就是用单引号括起的一个字符，例如`'a'`和`'0'`。字符文字量的类型是`char`。这样的字符文字量实际上是一种符号常量，表示在C++程序运行的计算机上的字符集中该字符的整数值。例如，如果你在一台使用ASCII字符集的机器上运行程序，`'0'`的值就是48。采用字符文字量而不用十进制写法能使程序更具可移植性。另有几个字符具有标准的名字，采用反斜线字符作为换意字符。例如，`\n`是换行符，`\t`是水平制表符。参看C.3.2节里有关各转义字符的细节。

宽字符文字量的形式是`L'ab'`，这里放在引号间的字符个数及意义由实现根据`wchar_t`类型确定。宽字符文字量的类型是`wchar_t`。

4.4 整数类型

与`char`一样，每个整数类型也有三种形式：“普通的”`int`，`signed int`和`unsigned int`。此外，整数还有三种大小：`short int`，“普通的”`int`和`long int`。`long int`可以简单地写成`long`。类似地，`short`也是`short int`的同义词。`unsigned`和`signed`分别是`unsigned int`和`signed int`的同义词。

`unsigned`整数类型对于将存储看做是二进制位数组的使用方式非常理想。采用`unsigned`而不用`int`以便多获得一个位去表示正整数，就不是什么好主意。通过将变量声明为`unsigned`而保证某些值始终为正的企图常常会被隐含的类型转换规则破坏（C.6.1节、C.6.2节）。

与`char`不同的是，普通的`int`总是有符号的。因此，那些有符号的`int`类型只不过是所对应的普通`int`类型的一个同义词罢了。

4.4.1 整数文字量

整数文字量有四种表面形式：十进制、八进制、十六进制和字符文字量（A.3节）。十进

制文字量用得最多，其形式如你所预期的

```
7 1234 976 1234567890 1234567890
```

编译器应该能够对文字量过长，无法表示的情况给出警告。

由0开头后跟 x ($0x$) 的文字量是十六进制数 (以16作为数的基数)，以0开头的文字量后面没有 x 的是八进制数 (以8为基数)。例如，

<i>decimal</i> :		2	63	83
<i>octal</i> :	0	02	077	0123
<i>hexadecimal</i> :	0x0	0x2	0x3f	0x53

字符 a 、 b 、 c 、 d 、 e 、 f 与其大写形式等价，分别表示10、11、12、13、14、15。八进制和十六进制形式特别适合用于表示二进制位的模式。使用这些记法表示真正的数值则常常会使人感到意外。例如，在一台将 int 表示为16位二补码整数的机器上， $0xffff$ 将是十进制数-1。如果表示整数所采用的位数更多，它就会是65535。

后缀 U 可以用于显式地写出 $unsigned$ 文字量。类似地，后缀 L 可用于显式地写 $long$ 文字量。例如，3是一个 int 型， $3U$ 是一个 $unsigned int$ 型，而 $3L$ 是一个 $long int$ 型。如果没有后缀，编译器就会基于文字量值的大小以及实现中整数的大小，为整数文字量确定一个适当的类型 (C.4节)。

限制使用那些意义不很明显的常量，只将它们用在给几个 $const$ (5.4节) 或者枚举量 (4.8节) 初始化的表示中，这是一种很好的做法。

4.5 浮点类型

浮点类型表示浮点数。像整数一样，浮点数也有三种大小： $float$ (单精度)， $double$ (双精度) 和 $long double$ (扩展精度)。

单、双和扩展精度的确切意义由实现确定。要想为某些正确选择数的精度具有重要意义的问题选择一种正确的精度，就需要有对浮点计算的深入理解。如果你并没有这种理解，那么给你一个忠告是花时间去学习，或者就选择 $double$ 并期待着能得到最好的结果。

4.5.1 浮点文字量

按默认规定，浮点文字量的类型是 $double$ 。再说一次，编译器应该能对浮点文字量太大以至无法表示的问题提出警告。下面是一些浮点文字量：

```
1.23 .23 0.23 1. 1.0 1.2e10 1.23e-15
```

请注意，在浮点文字量的中间不能出现空格。例如， $65.43 e-21$ 不是一个浮点文字量，而是下面这样的四个词法单词 (且将引起语法错误)：

```
65.43 e - 21
```

如果需要写类型为 $float$ 的浮点文字量，你可以通过后缀 f 或 F 定义它们：

```
3.14159265f 2.0f 2.997925F 2.9e-3f
```

如果需要类型为 $long double$ 的浮点文字量，你可以通过后缀 l 或 L 定义它们：

```
3.14159265L 2.0L 2.997925L 2.9e-3L
```


4.6 大小

C++ 基本类型的某些方面是由实现确定的，例如`int`的大小（C.2节）。我总要指出这种依赖性，并常常提出应该避免它们，或者通过某些方式尽可能减小其影响的建议。为什么需要为这些东西操心呢？在各种各样的系统中或使用多种编译器去编程序的人们很注意这些事情，因为如果他们不这样做，那他们就会被迫去花时间寻找和纠正很隐蔽的错误。那些自称不关心移植性的人也确实常常像自己所说的那样去做，因为他们只使用一种系统，并认为自己能承担起如下看法：“这个语言不过就是我的编译器所实现的那种东西”。但是，这实际上是一种狭隘和短视的观点。如果你的程序是成功的，那么它就很可能需要移植，而这时某些人就必须去寻找并纠正那些依赖于实现的特征了。此外，程序经常需要为了同一个系统而用其他编译器来编译，甚至你最喜爱的编译器的未来版本在做某些事情时，也可能采用与目前的版本不同的方式。在写程序时，理解程序对实现的依赖性的影响并予以限制，比后再去踩这个泥潭要容易得多。

限制依赖于实现的语言特征的影响并不太难，限制依赖于系统的库功能的影响就困难得多了。在所有可能之处都使用标准库的功能是一个办法。

提供了多种整数类型、多种无符号类型、多种浮点类型的原因就是希望使程序员能够利用各种硬件特性。在许多机器上，不同种类的基础类型之间，在对存储的需求、存储访问时间和计算速度方面存在着明显的差异。如果你了解一台机器，那么就很容易做出选择，例如，选择对某个变量所适用的整数类型。而写出真正可移植的低级代码就要困难得多。

C++对象的大小是用`char`的大小的倍数表示的，所以，按照定义`char`的大小为1。一个对象或者类型的大小可以用`sizeof`运算符得到（6.2节）。下面是基本类型的大小所能够保证的性质：

$$1 \equiv \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$$

$$1 \leq \text{sizeof}(\text{bool}) \leq \text{sizeof}(\text{long})$$

$$\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{wchar}_t) \leq \text{sizeof}(\text{long})$$

$$\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})$$

$$\text{sizeof}(N) \equiv \text{sizeof}(\text{signed } N) \equiv \text{sizeof}(\text{unsigned } N)$$

其中的 N 可以是`char`、`short int`、`int`或`long int`。此外，这里还保证`char`至少有8位，`short`至少有16位，而`long`至少有32位。一个`char`能保存机器的字符集中的一个字符。

这里是某个可能的基本类型集合和一个字符串的情况的图形表示：

char:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">a</div>
bool:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">1</div>
short:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">756</div>
int:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">100000000</div>
int*:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">&c1</div>
double:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">1234567e34</div>
char[14]:	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Hello, world!\0</div>

按照同样的尺度（一个字节0.2 in, 1 in = 0.0254 m），一兆字节的存储器将向右延展大约3 mile（5km, 1 mile = 1 609.344 m）。

char类型被假定是由实现选择的，在给定的计算机上选定最适合存储和操作字符的类型，典型情况下就是8位的字节。类似地，**int**类型也被假定是由实现选择的，采用在给定的计算机上最适合存储和操作整数的类型，典型情况下是一个4字节（32位）的机器字。假定得更多就不明智了，因为确实存在着某些采用32位字符的机器。

如果需要的话，有关某个具体实现的所有依赖于实现的特征都可以在<limits>里找到（22.2节）。例如，

```
#include <limits>
#include <iostream>
int main()
{
    cout << "largest float == " << numeric_limits<float>::max()
        << ", char is signed == " << numeric_limits<char>::is_signed << '\n';
}
```

在赋值和表达式里，都可以随意地混合使用各种基本类型。只要可能，有关的值将会被转换，尽可能不损失信息（C.6节）。

如果一个值*v*可以在一个类型*T*的变量里确切地表示，那么从*v*到*T*的转换将是保值的，不会有任何问题。在那些转换不能保值的地方最好是避免有关的转换（C.6.2.6节）。

你需要在某种细节程度上理解隐式转换，以便能完成重要的项目，特别是去理解别人写的实际代码。当然，这种理解对于阅读下面的各章并不是必需的。

4.7 void

类型**void**是一个语法上的基本类型。它可以作为更复杂的类型中的组成部分，但是没有类型为**void**的对象。**void**被用于刻画一个函数并不返回值，它还被用做指向不明类型的对象的指针的基础类型。例如，

```
void x;           // 错误：没有void对象
void f();         // 函数f不返回值（7.3节）
void* pv;         // 指向类型不明的对象的指针（5.6节）
```

在声明函数时，你必须刻画返回值的类型。逻辑上说，你可能希望通过忽略返回值类型来表示一个函数不返回值。但那样做将会减弱语法（附录A）的规范性，而且也与C的使用方式冲突。因此这里将**void**用做一个“伪返回类型”，用于说明一个函数不返回值。

4.8 枚举

一个枚举是一个类型，它可以保存一组由用户刻画的值。一旦定义之后，枚举的使用就很像是一个整数类型。

命名的整数常量可以定义为枚举的成员。例如，

```
enum { ASM, AUTO, BREAK };
```

这就定义了三个被称为枚举符的整数常量，并给它们赋了值。按默认方式，枚举符所赋的值从0开始递增，所以**ASM == 0**，**AUTO == 1**，**BREAK == 2**。枚举也可以命名，例如，

```
enum keyword { ASM, AUTO, BREAK };
```

每个枚举都是一个独立的类型，枚举符的类型就是它所在的那个枚举。例如，*AUTO*的类型就是*keyword*。

将一个变量声明为*keyword*而不是一个普通的*int*，能够给用户和编译器一些有关该变量拟议中的用途的提示。例如，

```
void f(keyword key)
{
    switch (key) {
        case ASM:
            // 做某些事情
            break;
        case BREAK:
            // 做某些事情
            break;
    }
}
```

编译器有可能提出一个警告，因为在三个*keyword*值中只有两个被处理了。

枚举符也可以用整型（4.1.1节）的*constant-expression*（常量表达式）（C.5节）进行初始化。如果某个枚举中所有枚举符的值均非负，该枚举的表示范围就是 $[0 : 2^k - 1]$ ，其中的 2^k 是能使所有枚举符位于此范围内的最小的2的幂；如果存在负的枚举符值，该枚举的取值范围就是 $[-2^k : 2^k - 1]$ 。这样就定义了一个最小的位段，其中能保存所有枚举符值的常规2补码表示，例如，

```
enum e1 { dark, light };           // 范围0:1
enum e2 { a = 3, b = 9 };          // 范围0:15
enum e3 { min = -10, max = 1000000 }; // 范围 -1048576:1048575
```

一个整型值可以显式地转换到一个枚举值。除非这种转换的结果位于该枚举的范围之内，否则就是无定义的。例如，

```
enum flag { x=1, y=2, z=4, e=8 }; // 范围0:15

flag f1 = 5;           // 类型错：5不是flag类型
flag f2 = flag(5);     // 可以：flag(5) 是flag类型且在flag的范围之内
flag f3 = flag(z|e);   // 可以：flag(12) 是flag类型且在flag的范围之内
flag f4 = flag(99);    // 无定义：99不在flag的范围之内
```

最后一个赋值说明了为什么不允许隐式地从整数转换到枚举：大部分整数值在特定的枚举里都没有对应的表示。

有关枚举的值范围的概念与Pascal一族语言中的枚举概念不同。无论如何，有关按位操作的各种例子在C和C++里已经有很长的历史了，其中都要求对超出枚举符集合的值有良好的定义。

一个枚举的*sizeof*就是某个能够容纳其范围的整型的*sizeof*，而且不会大于*sizeof(int)*，除非有某个枚举符的值不能用*int*也不能用*unsigned int*表示。举例来说，在*sizeof(int) == 4*的机器上，*sizeof(e1)*可以是1，也可能是4，但绝不会是8。

按照默认方式，枚举可以转换到整数去参加算术运算（6.2节）。一个枚举是一个用户定义类型，所以用户可以为枚举定义它自身的操作，例如定义++或<<（11.2.3节）。

4.9 声明

一个名字（标识符）能够在C++ 程序里使用之前必须首先声明。也就是说，必须刻画清楚它的类型，以通知编译器这个名字所引用的是哪一类的实体。这里有一些例子，展示了各种各样的声明：

```
char ch;
string s;
int count = 1;
const double pi = 3.1415926535897932385;
extern int error_number;

const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };

struct Date { int d, m, y; };
int day(Date* p) { return p->d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? -a : a; }

typedef complex<short> Point;
struct User;
enum Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

正如在这些例子中所看到的，在声明能做的事情可以比简单地为一个名字关联一个类型更多一些。这些声明中的大部分同时也是定义，也就是说，它们也定义了有关的名字所引用的那个实体。对于`ch`，它所对应实体就是适当数量的存储，以使它能够被用做一个变量——这块存储将被分配。`day`被定义的东西就是这里所描述的函数。常数`pi`被定义为具有值3.1415926535897932385。`Date`被定义为一个新类型。`Point`被定义就是类型`complex<short>`，所以`Point`也就成为`complex<short>`的同义词。在上面这些声明中，只有

```
double sqrt(double);
extern int error_number;
struct User;
```

不是定义，也就是说，它们所引用的实体必须在其他地方定义。函数`sqrt`的代码（体）必须通过另外的某个声明描述，`int`变量`error_number`的存储必须由某个另外的`error_number`的声明去分配，必须有另外的某个对类型`User`的声明定义出这个类型是什么样子。例如，

```
double sqrt(double d) { /* ... */ }
int error_number = 1;

struct User { /* ... */ };
```

在一个C++ 程序里（关于`#include`的影响，9.2.3节），每个命名实体必须有恰好一个定义。当然，它们可以有許多声明。一个实体的所有声明必须在所引用的类型上完全一致。所以，下面这个片段就包含了两个错误：

```
int count;
int count; // 错误：重复定义

extern int error_number;
extern short error_number; // 错误：类型不匹配
```

下面这个片段则没有错误（有关**extern**的使用参见9.2节）：

```
extern int error_number;
extern int error_number;
```

有些定义还为它们所定义的实体确定了一个“值”。例如，

```
struct Date { int d, m, y; };
typedef complex<short> Point;
int day(Date* p) { return p->d; }
const double pi = 3.1415926535897932385;
```

对于类型、模板、函数和常数，这种所谓的“值”是持久不变的。而对那些不是常量的数据类型^①，这种初始值可以在以后改变。例如，

```
void f()
{
    int count = 1;
    const char* name = "Bjarne"; // name是一个指向常量的变量（5.4.1节）
    // ..
    count = 2;
    name = "Marian";
}
```

在前面的定义里，只有

```
char ch;
string s;
```

没有描述有关的值。对于如何以及何时给变量赋以默认值的问题，请参看4.9.5节和10.4.2节的解释。任何描述了初始值的声明都是一个定义。

4.9.1 声明的结构

一个声明由四个部分组成：一个可选的“描述符”，一个基础类型，一个声明符，还有一个可选的初始式。除了函数和名字空间之外，其他声明都应该由分号结束。例如，

```
char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

这里的基础类型是**char**，声明符是***kings[]**，而初始式是**= { ... }**。

描述符是一个开始关键字，例如**virtual**（2.5.5节、12.2.6节）和**extern**（9.2节），它们说明了被声明事物的某些非类型的属性。

声明符由一个名字和可选的若干声明运算符组成。最常用的声明运算符是（A.7.1节）

*	指针	前缀
*const	常量指针	前缀
&	引用	前缀
[]	数组	后缀
()	函数	后缀

如果这些东西都是前缀或者都是后缀，其使用就会很简单。但是，*、[]和()被设计为模仿它们在表达式（6.2节）里的使用方式。这样，*就是前缀的，而[]和()都是后缀的。而且

① 原书如此，实际指的是这些类型的“数据对象”。——译者注

后缀的声明运算符比前缀的那些声明运算符约束力更强。因此，**kings[]* 就是一个指向什么东西的（某个）指针数组。还有，我们有时就必须使用括号，例如要表示“指向函数的指针”，见5.1节的例子。完整的细节请看附录A。

请注意，在一个声明中不能没有类型。例如，

```
const c = 7;    // 错误：无类型
gt(int a, int b) { return (a>b) ? a : b; } // 错误：无返回类型

unsigned ui;    // 可以：'unsigned' 就是 'unsigned int'
long li;        // 可以：'long' 就是 'long int'
```

在这里，标准C++与C和C++的早期版本有所不同，原来一直将前面两个例子看成以*int*作为没有明确说明的类型（B.2节）。这种“隐式的*int*”是许多微妙错误和混乱的一个根源。

4.9.2 声明多个名字

可以在单个的声明中声明多个名字。这种声明包含一个由逗号分隔的声明符的表列。例如，我们可以像下面这样声明两个整数：

```
int x, y;        // int x; int y;
```

注意，运算符只作用于一个单独的名字——而不是同一声明中随后写的所有名字。例如，

```
int* p, y;        // int* p; int y; 不是 int* y;
int x, *q;        // int x; int* q;
int v[10], *pv;   // int v[10]; int* pv;
```

这种结构不利于阅读程序，因此应该避免。

4.9.3 名字

一个名字（标识符）由一系列字母和数字构成，其中的第一个字符必须是字母。下划线字符也作为字母看待。C++ 对一个名字里字符的个数并未强加任何限制。然而，一个具体实现中的某些部分可能不受写编译器的人们控制（特别是连接器），而这些部分有可能强加某些限制，这当然非常不幸。有些运行环境还对标识符中可以接受的字符做了某些扩充或者限制。扩充（例如，允许在名字中使用\$）将产生不可移植的程序。C++ 的关键字（附录A），例如，*new*和*int*，不能用做用户定义实体的名字。名字的例子如下所示：

<u>hello</u>	<u>this_is_a_most_unusually_long_name</u>			
DEFINED	foO	bAr	u_name	HorseSense
var0	var1	CLASS	_class	___

不能作为标识符的字符序列的例子如：

012	a fool	\$sys	class	3var
pay.due	foo~bar	.name	if	

以下划线符开头的名字是保留给实现或者运行环境，用于特殊功能的，在应用程序里不要采用这样的名字。

编译器在读程序时，总是设法去寻找最长的能够做成一个名字的字符序列。这样，*var10*就是一个名字，而不是名字*var*后面跟着数*10*。还有，*elseif*是一个名字，而不是关键字*else*后跟关键字*if*。

大写和小写字母是区分的，所以`Count`和`count`是不同的名字，选择那些只是在大小写上有差异的名字可不是一种聪明的做法。一般来说，最好能避免那些仅以某些微细的方式区分的不同的名字。举例来说，大写的`o` (`O`) 和零 (`0`) 很难分辨，小写的`L` (`l`) 和 `-` (`1`) 也是这样。因此，用`IO`、`lo`、`ll`、`ll`作为标识符名字是很糟糕的选择。

用于较大的作用域的名字应该是相对比较长的更加明确的名字，例如`vector`、`Window_with_border`和`Department_number`。然而，如果在很小的作用域里只使用那些短小而熟悉的名字，如`x`、`i`和`p`，代码会显得更清晰些。类（第10章）和名字空间（8.2节）可用来保持较小的作用域。让那些频繁使用的名字相对较短，将较长的名字保留给不常用的实体，这种做法也很有价值。名字的选择应该反映一个实体的意义，而不是它的实现。例如，`phone_book`就比`number_list`好，即使这些电话号码实际存放在一个`list`（3.7节）里。选择好的名字也是一种艺术。

应设法保持一种统一的命名风格。例如，对非标准库的用户定义类型都用大写，非类型的开头用小写字母（例如，`Shape`和`current_token`）。还有，对宏用全部大写的名字（如果你真的要使用宏的话；例如，`HACK`），并用下划线分隔名字中的单词。当然，统一用法是很难做到的，因为程序常常由一些片段组成，它们有不同的来源，采用的是不同的也都合理的风格。你自己对缩写和首字母缩写应该保持某种一致性。

4.9.4 作用域

一个声明将一个名字引进一个作用域；也就是说，这个名字只能在程序正文的一个特定部分内使用。对于在函数里声明的名字（经常被称为局部名字），其作用域从它声明的那一点开始，直到这个声明所在的块结束为止。一个块就是由`{ }`围起的一段代码。

一个名字称为是全局的，如果它是在所有函数、类（第10章）和名字空间（8.2节）之外定义的。全局名字的作用域从声明的那一点开始，一直延伸到这个声明所在文件的结束。在一个块里声明的名字可以遮蔽在其外围的块里所声明的名字或者全局的名字。也就是说，在一个块里可以重新定义一个名字，让它去引用另一个不同的实体。在退出这个块之后，该名字又恢复了它原来的意义。例如，

```
int x;           // 全局的x

void f()
{
    int x;       // 局部的x遮蔽了全局的x
    x = 1;       // 给局部的x赋值

    {
        int x;   // 遮蔽第一个局部的x
        x = 2;   // 给第二个局部的x赋值
    }

    x = 3;       // 给第一个局部的x赋值
}

int* p = &x;     // 取全局x的地址
```

遮蔽某些名字的情况在写大程序时是不可避免的。但是，读程序的人很容易没有注意某个名字已经被遮蔽了。由于这类错误相对不那么常见，它们反而很难被发现。因此，还是应该尽

量避免名字遮蔽的情况。对全局变量或者很大的函数里的局部变量，使用像*i*或*x*一类的名字实际上就是自找麻烦。

被遮蔽的全局名字可以通过作用域解析运算符 `::` 去引用。例如，

```
int x;

void f2()
{
    int x = 1; // 遮蔽全局的x
    ::x = 2;   // 给全局的x赋值
    x = 2;     // 给局部的x赋值
    // ...
}
```

没有办法去使用被遮蔽的局部名字。

一个名字的作用域从它被声明的那点开始；也就是说，在声明符结束之后，初始式的开始之前。这意味着一个名字甚至可以用于描述它自己的初始值。例如，

```
int x;
void f3()
{
    int x = x; // 不当：用x自己（未初始化）的值初始化x。
}
```

这样做并不非法，只是荒谬。好编译器能对变量在未设置之前就使用提出警告（5.9[9]）。

在一个块里，也有可能同一个名字（不通过 `::` 运算符）实际引用的是两个不同的对象。看下面例子：

```
int x = 11;

void f4()          // 不当：
{
    int y = x;      // 使用全局的x: y = 11
    int x = 22;
    y = x;          // 使用局部的x: y = 22
}
```

函数参数被当做在函数最外层的块中声明的名字。所以

```
void f5(int x)
{
    int x;          // 错误
}
```

是错误的，因为*x*在同一个作用域里定义了两次。将这种情况作为错误，将能捕捉到一种相当常见的微妙失误。

4.9.5 初始化

如果为一个对象提供了初始式，这个初始式将确定对象的初始值。如果没有提供初始式，全局的（4.9.4节）、名字空间的（8.2节）和局部静态的（7.1.2节、10.2.4节）对象（统称为静态对象）将被自动初始化为适当类型的0。例如，

```
int a;           // 意思是 "int a = 0;"
double d;        // 意思是 "double d = 0.0;"
```


局部对象（有时称为自动对象）和在自由存储区里建立的对象（有时被称为动态对象或者堆对象）将不会用默认值做初始化。例如，

```
void f()
{
    int x;    // x没有定义良好的值
    // ...
}
```

数组和结构的成员，也根据数组或结构是否为静态来确定是否默认地进行初始化。用户定义类型可以有自定义的默认初始化方式（10.4.2节）。

更复杂的对象需要以多于一个的值作为初始式。数组（5.2.1节）和结构（5.7节）的C风格初始化采用的是 { 和 } 括起的初始式列表描述。带有构造函数的用户定义类型采用的是函数风格的参数表形式（2.5.2节、10.2.3节）。

请注意，在声明中写一对空的括号总意味着是“函数”（7.1节）。例如，

```
int a[] = { 1, 2 };    // 数组初始式
Point z(1, 2);        // 函数风格的初始式（通过构造函数完成初始化）
int f();              // 函数声明
```

4.9.6 对象和左值

我们可以分配和使用没有名字的“变量”，而且可能对一个看起来很奇怪的表达式赋值（例如， $*p[a + 10] = 7$ ）。因此，对一个名字的某种需要就是它应该表示“存储器里的什么东西”。这也就是最简单最基本的对象概念。这样，一个对象就是存储中一片连续的区域；左值就是引用某个对象的表达式。术语左值原本是想说“某个可以放在赋值左边的东西”。然而，并不是每个左值都能够被用在赋值的左边，左值也可以是引用了某个常量（5.5节）。没有被声明为常量的左值常常被称做是可修改的左值。不要将对象的这种简单和低级的概念与类对象和多态性类型（15.4.3节）的对象概念混淆了。

除非程序员另有描述（7.1.2节、10.4.8节），在一个函数里声明的对象都在其定义被遇到之时建立，在它的名字离开作用域的时候被销毁（10.4.4节）。这种对象被称做自动对象。在全局和名字空间作用域里的对象，以及在函数和类里声明为`static`的对象只建立一次（仅仅一次），它们一直“生存”到程序结束（10.4.9节）。这种对象被称为静态对象。数组成员、非静态结构和类的成员的生存期由它们作为其部分的那些对象决定。

通过`new`和`delete`运算符，你可以建立生存期可以直接控制的对象（6.2.6节）。

4.9.7 typedef

如果一个声明以`typedef`为前缀，它就是为类型声明了一个新名字，而不是声明一个指定类型的对象。例如，

```
typedef char* Pchar;
Pchar p1, p2;    // p1和p2都是char*s
char* p3 = p1;
```

一个这样定义的名字称为一个“`typedef`”，可以方便地作为原来较笨拙的类型名的缩写。例如，`unsigned char`对于频繁使用而言显得太长，那么我们就可以为之定义一个同义词：

```
typedef unsigned char uchar;
```

typedef的另一类使用是将对某个类型的直接引用限制到一个地方。例如，

```
typedef int int32;
typedef short int16;
```

如果我们在所有需要可能较大的整数的地方都用**int32**，那么就很容易将我们的程序移植到一个**sizeof(int)**是2的机器上，因为只要在代码中重新定义**int32**的惟一一个出现

```
typedef long int32;
```

无论是好是坏，**typedef**都只是其他类型的同义词，而不是独立的类型。因此，**typedef**可以随意地与作为其同义词的类型混合使用。如果希望得到独立的类型，而又有着相同的语义或相同的表示，那么就应该去用枚举（4.8节）或者类（第10章）。

4.10 忠告

- [1] 保持较小的作用域；4.9.4节。
- [2] 不要在一个作用域和它外围的作用域里采用同样的名字；4.9.4节。
- [3] 在一个声明中（只）声明一个名字；4.9.2节。
- [4] 让常用的和局部的名字比较短，让不常用的和全局的名字比较长；4.9.3节。
- [5] 避免看起来类似的名字；4.9.3节。
- [6] 维持某种统一的命名风格；4.9.3节。
- [7] 仔细选择名字，反映其意义而不是反映实现方式；4.9.3节；
- [8] 如果所用的内部类型表示某种可能变化的值，请用**typedef**为它定义一个有意义的名字；4.9.7节。
- [9] 用**typedef**为类型定义同义词，用枚举或类去定义新类型；4.9.7节。
- [10] 切记每个声明中都必须描述一个类型（没有“隐式的**int**”）；4.9.1节。
- [11] 避免有关字符数值的不必要假设；4.3.1节、C.6.2.1节。
- [12] 避免有关整数大小的不必要假设；4.6节。
- [13] 避免有关浮点类型表示范围的不必要假设；4.6节。
- [14] 优先使用普通的**int**而不是**short int**或者**long int**；4.6节。
- [15] 优先使用**double**而不是**float**或者**long double**；4.5节。
- [16] 优先使用普通的**char**而不是**signed char**或者**unsigned char**；C.3.4节。
- [17] 避免做出有关对象大小的不必要假设；4.6节。
- [18] 避免无符号算术；4.4节。
- [19] 应该带着疑问去看待从**signed**到**unsigned**，或者从**unsigned**到**signed**的转换；C.6.2.6节。
- [20] 应该带着疑问去看待从浮点到整数的转换；C.6.2.6节。
- [21] 应该带着疑问去看待向较小类型的转换，如将**int**转换到**char**；C.6.2.6节。

4.11 练习

- 1. (*2) 让“Hello, world!”程序（3.2节）运行。如果程序无法按所写的形式运行，请看B.3.1节。
- 2. (*1) 对于4.9节的每个声明做下面事情：如果该声明不是一个定义，请为它写一个定义。如果该声明是一个定义，请改写，使它成为不是定义的声明。

3. (*1.5) 写一个程序打印出各种基本类型、几个指针类型和几个你所选择的枚举类型的大小。使用`sizeof`运算符。
4. (*1.5) 写一个程序打印出字母 'a' .. 'z' 和数字 '0' .. '9'，以及它们的整数值。对所有其他可打印字符做同样的事情。再用16进制形式做同样的事情。
5. (*2) 在你所用的机器上，下面类型的最大值和最小值是什么：*char*、*short*、*int*、*long*、*float*、*double*、*long double*和*unsigned*。
6. (*1) 什么是可以在你系统上的C++程序里使用的最长的局部名字？什么是可以在你系统上的C++程序里使用的最长的外部名字？对于你在名字中能够使用的字符有任何限制吗？
7. (*2) 为整数和基本类型画一张图，其中一个类型指向另一个类型，如果在符合标准的实现中，第一个类型可以表示的所有值都可以在第二个类型中表示。为你所喜爱的实现画出另一张图。

第5章 指针、数组和结构

崇高与荒谬

常常如此紧密相关，
以至要区分它们都非常困难。

——Tom Paine

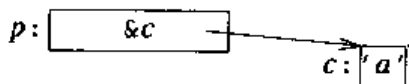
指针——零——数组——字符串文字量——到数组的指针——常量——指针和常量——
引用——*void**——数据结构——忠告——练习

5.1 指针

对类型 T ， T^* 是“到 T 的指针”类型，也就是说，一个类型为 T^* 的变量能保存一个类型 T 的对象的地址。例如，

```
char c = 'a';  
char* p = &c;    // p保存着c的地址
```

用图表示是



不幸的是，到数组的指针和到函数的指针需要更复杂的记法：

```
int* pi;           // 到int的指针  
char** ppc;        // 到字符的指针的指针  
int* ap[15];       // 15个到int的指针的数组  
int (*fp)(char*);  // 到函数的指针，这种函数以char* 为参数、返回int  
int* f(char*);     // 有一个char* 参数的函数，返回一个到int的指针
```

参看4.9.1节有关声明语法的解释，以及附录A中的完整语法。

对指针的基本操作是间接引用[⊖]，也就是说，引用被指针所指的那个对象。这一操作也被称做间接（indirection）。间接运算符是（前缀的）一元 $*$ 。例如，

```
char c = 'a';  
char* p = &c;    // p保存着c的地址  
char c2 = *p;    // c2 == 'a'
```

被 p 所指的变量是 c ，而 c 中所存的值是 $'a'$ 。所以由 $*p$ 赋给 $c2$ 的值就是 $'a'$ 。

对于指向数组元素的指针可以执行一些算术运算（5.3节）。指向函数的指针非常有用，有关问题将在7.7节介绍。

指针的实现是希望能直接映射到程序运行所在的机器上的地址机制。许多机器可以对字

[⊖] 英文词为dereference或其变形。这里将译为“间接”、“间接访问”或“间接引用”。——译者注

节寻址。那些不能这样做的机器多半都有能从机器字中取出字节的硬件功能。在另一方面，还有极少数的机器可以寻址到单个的二进制位。由于这些情况，能够独立地进行分配和通过内部指针类型指向的最小对象就是`char`。请注意，一个`bool`量最少也要占据像`char`那么大的空间（4.6节）。如果想更紧凑地存储更小的值，你可以使用按位逻辑操作（6.2.4节）或者结构中的位域（C.8.1节）。

5.1.1 零

零（0）是一个整数。由于各种标准转换（6.2.3节），0可以被用于作为任意整型（4.1.1节）、浮点类型、指针、还有指向成员的指针的常量。0的类型将由环境确定。在典型情况下0被表示为一个适当大小的全零二进制位的模式（但也不必如此）。

没有任何对象会被分配到地址0，因此，0也被当做一个指针文字量，表明一个指针当时并没有指向任何对象。

在C中流行的是用一个宏`NULL`表示0指针。由于C++ 收紧的类型检查规则，采用普通的0而不是一些人建议的`NULL`宏，带来的问题会更少一些。如果你感到必须定义`NULL`，请采用

```
const int NULL = 0;
```

用`const`限定词（5.4节）是防止无意地重新定义`NULL`，并保证`NULL`可以用到那些要求常量的地方。

5.2 数组

对于类型`T`，`T[size]` 就是“具有`size`个`T`类型的元素的数组”类型。这些元素的下标从0到`size - 1`。看下面例子：

```
float v[3];      // 一个数组，包含3个浮点数：v[0], v[1], v[2]
char* a[32];     // 一个数组，包含32个到char的指针：a[0]..a[31]
```

数组元素的个数，即数组的界，必须是一个常量表达式（C.5节）。如果你需要变化的界，那么请用`vector`（3.7.1节、16.3节）。例如，

```
void f(int i)
{
    int v1[i];      // 错误：数组大小必须是常量表达式
    vector<int> v2(i); // 可以
}
```

多维数组被表示为数组的数组。例如，

```
int d2[10][20]; // d2是一个包含10个各包含20个整数的数组的数组
```

其他语言对数组的界采用逗号记法，用在这里将产生一个编译时错误。因为逗号（,）是序列运算符（6.2.2节），不允许出现在常量表达式里。例如，请试试这个：

```
int bad[5,2];      // 错误：逗号不允许出现在常量表达式里
```

多维数组在C.7节描述。在底层代码中之外最好避免使用它们。

5.2.1 数组初始化

数组可以用一系列值进行初始化。例如，

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

当数组声明中没有给出数组大小，但是有初始式列表时，数组的大小就通过数出列表中元素个数的方式确定。因此，`v1`和`v2`的类型分别是`int[4]`和`char[4]`。如果明确给出了大小，在初始化列表中给了多余的元素就是错误。例如，

```
char v3[2] = { 'a', 'b', 0 }; // 错误：初始式太多
char v4[3] = { 'a', 'b', 0 }; // 可以
```

如果初始式列表里的元素太少，剩余的元素将被设定为0。例如，

```
int v5[8] = { 1, 2, 3, 4 };
```

等价于

```
int v5[] = { 1, 2, 3, 4, 0, 0, 0, 0 };
```

请注意，并不存在与数组初始化相对应的数组赋值：

```
void f()
{
    v4 = { 'c', 'd', 0 }; // 错误：没有数组赋值
}
```

如果你需要用这种赋值，请使用`vector`（16.3节）或者`valarray`（22.4节）。

字符的数组可以很方便地用字符串文字量（5.2.2节）进行初始化。

5.2.2 字符串文字量

字符串文字量是用双引号括起的字符序列：

```
"this is a string"
```

一个字符串文字量里包含的字符个数比它看起来的字符数多一个，它总由一个空字符 `'\0'` 结束。空字符的值是0。例如，

```
sizeof("Bohr") == 5
```

字符串文字量的类型是“适当个数的`const`字符的数组”，所以 `"Bohr"` 的类型就是`const char[5]`。

可以用字符串文字量给一个`char*` 赋值。允许这样做是因为在C和C++原来的定义里，字符串文字量的类型就是`char*`。允许将字符串文字量赋值给`char*` 保证了成百万行的C和C++程序继续为合法程序。但试图通过这样的指针去修改字符串文字量将是一个错误：

```
void f()
{
    char* p = "Plato";
    p[4] = 'e'; // 错误：给常量赋值；结果无定义
}
```

一般来说，这类错误未必能在运行之前捕捉到，而各种实现在对本规则的强调方面也会很不相同。参见B.2.3节。将字符串文字量当做常量，不仅因为这样做很明显，而且它也能允许实现对于字符串文字量的存储和访问做一些效果很显著的优化。

如果我们希望一个字符串保证能够被修改，那么就必须将有关的字符复制到数组里：

```
void f()
{
    char p[] = "Zeno";    // p是5个char的数组
    p[0] = 'R';          // 可以
}
```

字符串文字量是静态分配的，所以让函数返回它们是安全的。例如，

```
const char* error_message(int i)
{
    // ...
    return "range error";
}
```

保存 "range error" 的存储区在 `error_message()` 的调用之后并不会丢掉。

两个同样的字符串文字量是否被分配在一起，这一点由实现确定 (C.1节)。例如，

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";
void g()
{
    if (p == q) cout << "one!\n"; // 结果是由实现确定的
    // ...
}
```

请注意，在应用于指针时，`==` 比较的是地址（指针的值），而不是被指的东西。

空字符串写成一对连续的引号 ""，它的类型是 `const char[1]`。

为表示非打印字符而提供的反斜线约定 (C.3.2节) 也可以用在字符串里。这就使我们可以 在字符串里表示诸如双引号 (") 和转义字符反斜线 (\) 等。这类字符中最常见的就是换行符 '\n'。例如，

```
cout << "beep at end of message\n";
```

转义字符 '\a' 是 ASCII 字符 BEL（也被称做警铃），它会导致发出某种声音。

在字符串里不能有“真正的”换行：

```
"this is not a string
but a syntax error"
```

为使程序比较整洁，可以将长字符串通过空白字符断开[⊖]。例如，

```
char alpha[] = "abcdefghijklmnopqrstuvwxy"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

编译器将拼接起连续的字符串，所以对 `alpha` 可以等价地用下面这个字符串来做初始化：

```
"abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

将空字符放到字符串里也是可能的，但是大部分程序将不会去考虑在这个字符的后面还有其他字符。举例来说，字符串 "Jers\000Munk" 将被各种标准库函数，例如 `strcpy()` 和 `strlen()`，当做 "Jers" 看待；参见 20.4.1 节。

带有前缀 `L` 的字符串，例如 `L"angst"`，是宽字符的字符串（4.3 节、C.3.3 节），其类型是 `const wchar_t[]`。

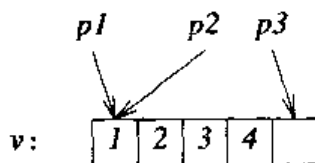
⊖ 请注意，换行符、制表符也都是空白字符。——译者注

5.3 到数组的指针

在C++ 里，指针和数组密切相关。一个数组的名字能够被用做到它的开始元素的指针。例如，

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;           // 指向开始元素（隐式转换）
int* p2 = &v[0];       // 指向开始元素
int* p3 = &v[4];       // 指向最后元素之后一个位置
```

其图示是，



取得超出一个数组结束之后一个元素位置的指针，这件事将保证可以做到。这一点在许多算法中都非常重要（2.7.2节、18.3节）。当然，这个指针事实上并不指向数组里的一个元素，因此不能通过它去读或者写。取得数组开始元素之前的元素地址是无定义的，应该避免。在某些机器结构中，数组常常被分配在机器地址的边界上，所以“开始元素之前的一个位置”根本就没有意义。

从数组名到这个数组的开始元素的指针的隐式转换，在C风格代码的函数调用中广泛使用。例如，

```
extern "C" int strlen(const char*); // 来自<string.h>

void f()
{
    char v[] = "Annemarie";
    char* p = v;           // 隐式地从char[]转换到char*
    strlen(p);
    strlen(v);             // 隐式地从char[]转换到char*
    v = p;                 // 错误：不能给数组赋值
}
```

在两个调用中，传给标准库函数`strlen()`将是同一个值。困难在于无法避免这种隐式的转换。换句话说，没办法去定义一个函数，使得在函数调用时能够将整个数组复制给它。幸好并不存在从指针到数组的隐式转换。

数组参数被隐式地转换到指针，这也意味着对于被调用函数而言，数组的大小就会被丢掉。这就要求被调函数必须以某种方式去确定数组的大小，以便完成各种有意义的操作。与C标准库中的其他以字符指针为参数的函数一样，`strlen()`依靠0确定字符串结束。`strlen(p)`将返回直到表示结束位置的0（但不包括它）的字符个数。这是一类相当低级的东西。标准库`vector`（16.3节）和`string`（第20章）都不会遇到这种问题。

5.3.1 在数组里漫游

高效而优雅地访问数组（以及其他类似数据结构）是许多算法（3.8节、第18章）实现的关键。这种访问可以通过到数组的一个指针再加上一个下标，或者通过一个到数组元素的指

针进行。下面是采用下标遍历一个字符串的例子：

```
void fi(char v[])
{
    for (int i = 0; v[i] != 0; i++) use(v[i]);
}
```

这等价于下面通过指针遍历：

```
void fp(char v[])
{
    for (char* p = v; *p != 0; p++) use(*p);
}
```

前缀运算符 `*` 对指针做间接，所以 `*p` 就是 `p` 所指的字符；而 `++` 对指针做增量操作，使其索引这个数组中的下一个字符。

没有任何内在的理由说某种方式快于另一种。通过新型的编译器，这两个例子完全可能产生一模一样的代码（见5.9[8]）。程序员可以根据自己的逻辑或者美学观点选择其一。

将算术运算符 `+`、`-`、`++` 或 `--` 应用于指针的结果依赖于被指对象的类型。当某个算术运算符被应用于类型 `T*` 的指针 `p` 时，假定 `p` 是指向类型 `T` 的对象的数组中的一个元素；那么 `p + 1` 将指向下一个元素，而 `p - 1` 将指向前一个元素。这也意味着 `p + 1` 的整数值比 `p` 的整数值大 `sizeof(T)`。例如执行

```
#include <iostream>
int main ()
{
    int vi[10];
    short vs[10];

    std::cout << &vi[0] << ' ' << &vi[1] << '\n';
    std::cout << &vs[0] << ' ' << &vs[1] << '\n';
}
```

可能产生

```
0x7fffaef0 0x7fffaef4
0x7fffaedc 0x7fffaede
```

对指针值，默认方式采用的是16进制记法输出。上面这些是在我所用的实现上的情况，其中 `sizeof(short)` 是2而 `sizeof(int)` 是4。

只有在两个指针指向同一个数组的元素时，指针之间相减才有定义（虽然语言并没有一个严格的规则保证这一点）。如果从一个指针减去另一个指针，结果就是这两个指针之间的数组元素个数（一个整数）。在一个指针上加一个整数，或者减一个整数，得到的结果还是一个指针值。如果这个值并不指向原来那个指针所指的数组的元素或者是超出其末端一个位置，那么使用这个值产生的结果是无定义的。例如，

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5] - &v1[3]; // i1 = 2
    int i2 = &v1[5] - &v2[3]; // 结果无定义
```

```

int* p1 = v2+2;           // p1 = &v2[2]
int* p2 = v2-2;           // *p2无定义
}

```

复杂的指针算术通常并不必要，最好避免使用。指针相加没有任何意义，因此是不允许的。

数组不具有自描述性，因为并不保证与数组一起保存着这个数组的元素个数。这就意味着，如果要遍历一个数组，而该数组并不像字符串那样包含一个结束符，我们就必须以某种方式提供它的元素个数。例如，

```

void fp(char v[], unsigned int size)
{
    for (int i=0; i<size; i++) use(v[i]);

    const int N = 7;
    char v2[N];
    for (int i=0; i<N; i++) use(v2[i]);
}

```

请注意，大多数C++ 实现都不提供对数组范围的检查。这种数组的概念从本质上就是非常低级的，更高级的数组概念可以通过类的方式提供，参见3.7.1节。

5.4 常量

C++ 提供了用户定义常量的概念，**const**就是为了直接表述“不变化的值”这样一个概念。这种东西在一些环境中非常有用，例如，许多对象在初始化之后就不再改变自己的值了；与直接将文字量散布在代码中相比，采用符号常量写出的代码更容易维护；指针常常是边读边移动，而不是边写边移动；许多函数参数是只读不写的。

关键字**const**可以加到一个对象的声明上，将这个对象声明为一个常量。因为不允许赋值，常量就必须进行初始化。例如，

```

const int model = 90;           // model是个常量
const int v[] = { 1, 2, 3, 4 }; // v[i] 是常量
const int x;                    // 错误：没有初始化

```

将某些东西声明为常量，就保证了在其作用域内不能改变它们的值：

```

void f()
{
    model = 200; // 错误
    v[2]++;     // 错误
}

```

请注意，**const**实际上改变了类型，也就是说，它限制了对象能够使用的方式，并没有描述常量应该如何分配。例如，

```

void g(const X* p)
{
    // 在这里不能修改 *p
}

void h()
{
    X val; // val可以被修改
    g(&val);
}

```

```
// ...
}
```

编译器可以以多种方式利用一个对象是常量的这一性质，当然，这实际上要看它有多么聪明。例如，对于常量的初始式常常是（也并不总是）常量表达式（C.5节）；如果确实是这样，那么这个常量就可以在编译时进行求值。进一步说，如果编译器知道了某*const*的所有使用，它甚至可以不为该*const*分配空间。例如，

```
const int c1 = 1;
const int c2 = 2;
const int c3 = my_f(3); // 编译时不知道c3的值
extern const int c4;    // 编译时不知道c4的值
const int* p = &c2;     // 需要为c2分配空间
```

在这里，编译器能够知道*c1*和*c2*的值，所以就可以将这些值用到常量表达式里。由于在编译时无法知道*c3*和*c4*的值（仅仅通过位于这个编译单元中的信息，见9.1节），因此必须为*c3*和*c4*分配存储。又由于*c2*的地址被取用（而且应该假定它会在其他地方用），所以*c2*也需要分配存储。最简单的常见情况就是常量的值在编译时已知，而且不需要分配存储，*c1*就属于这种情况。关键字*extern*说明*c4*是在其他地方定义的（9.2节）。

对于常量的数组，典型的情况是需要分配存储，因为一般说编译器无法弄清楚表达式里使用的是数组中的哪些元素。但无论如何，在许多机器上，甚至对这种情况也可以通过把常量的数组放进只读存储器里，并因此得到效率的改善。

*const*的最常见用途是作为数组的界和作为分情况标号。例如，

```
const int a = 42;
const int b = 99;
const int max = 128;

int v[max];

void f(int i)
{
    switch (i) {
        case a:
            // ...
        case b:
            // ...
    }
}
```

对于这类情况，人们也常用枚举符（4.8节）来代替*const*。

关于将*const*用到类成员函数上的使用方式将在第10.2.6节和第10.2.7节讨论。

应该在程序里系统化地使用符号常量，以避免代码中“神秘的数值”。如果某个数值常量在代码中反复出现，例如某个数组的界，要修改这些代码就可能变得很麻烦，因为要完成一次正确的更新，该常量的每一次出现都必须修改。采用符号常量可以使信息局部化。常见情况是，一个数值常量代表着程序中的一个假设。例如，4可能代表一个整数的字节数，128可能代表需要缓冲的输入字符个数，6.24可能代表丹麦货币与美元的兑换比率。让这些数值常量散布到代码中，对程序的维护者而言这些值是很难辨认和理解的。常见的情况是，在程序移植时，或者当某些其他修改破坏了它们所表示的假设时，由于没有注意到这种数值，以致使它们变成了程序里的错误。将这种假设用加有良好注释的符号常量表示，就可以大大减少

这一类维护问题。

5.4.1 指针和常量

使用一个指针时涉及到两个对象：该指针本身和被它所指的对象。将一个指针的声明用 *const* “预先固定”将使那个对象而不是使这个指针成为常量。要将指针本身而不是被指对象声明为常量，我们必须使用声明运算符 **const*，而不能只用简单的 *const*。例如，

```
void f1(char* p)
{
    char s[] = "Gorm";

    const char* pc = s;           // 指向常量
    pc[3] = 'g';                  // 错误：pc是指向常量的
    pc = p;                       // ok

    char*const cp = s;            // 常量指针
    cp[3] = 'a';                  // ok
    cp = p;                       // 错误：cp是常量指针

    const char*const cpc = s;     // 到const的const指针
    cpc[3] = 'a';                 // 错误：cpc指向常量
    cpc = p;                      // 错误：cpc本身是常量
}
```

定义常量指针的声明运算符是 **const*。并没有 *const** 声明符，所以出现在 *** 之前的 *const* 是作为基础类型的一部分。例如，

```
char*const cp;    // 到char的const指针
char const* pc;   // 到const char的指针
const char* pc2;  // 到const char的指针
```

有人发现从右向左读这种定义很有帮助。例如，“*cp*是一个*const*指针到*char*”，以及“*pc2*是一个指针指到*const char*”^①。

一个某时通过指针访问当做常量的对象，也完全可能在以其他方式访问时被作为变量。对于函数参数而言，这个情况就特别有用。通过将指针参数声明为 *const*，就禁止了这个函数对被指参数的修改。例如，

```
char* strcpy(char* p, const char* q); // 不能修改 *q
```

你可以将一个变量的地址赋给一个到常量的指针，因为这样做不会造成任何伤害。当然，不能将常量的地址赋给一个未加限制的指针，因为这样将会允许修改该对象的值了。例如，

```
void f4()
{
    int a = 1;
    const int c = 2;
    const int* p1 = &c; // ok
    const int* p2 = &a; // ok
    int* p3 = &c;       // 错误：用const int* 对int* 进行初始化
    *p3 = 7;            // 试图修改c的值
}
```

① 这样从右向左的方式按中文读起来并不方便。这里是硬按顺序翻译的，以模仿原意。原文为“*cp* is a *const* pointer to *char*”和“*cp2* is a pointer to *const char*”。——译者注

可以通过对`const`指针的显式类型转换，明确要求去掉这种限制（10.2.7.1节和15.4.2.1节）。

5.5 引用

一个引用就是某对象的另一个名字。引用的主要用途是为了描述函数的参数和返回值，特别是为了运算符的重载（第11章）。记法`X&`表示到`X`的引用。例如，

```
void f()
{
    int i = 1;
    int& r = i;    // r和i现在引用同一个int
    int x = r;     // x = 1
    r = 2;         // i = 2
}
```

为了确保一个引用总能是某个东西的名字（也就是说，总能约束到某个对象），我们必须对引用做初始化。例如，

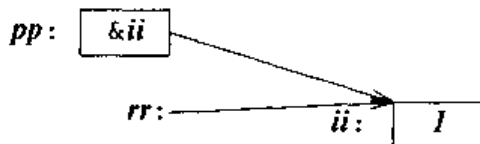
```
int i = 1;
int& r1 = i;    // 正确：r1被初始化
int& r2;        // 错误：没有初始化
extern int& r3;  // 正确：r3在别处初始化
```

对一个引用的初始化是与对它赋值完全不同的另一件事情。除了外表形式之外，实际上根本就没有能操作引用的运算符操作。例如，

```
void g()
{
    int ii = 0;
    int& rr = ii;
    rr++;        // ii被增加1
    int* pp = &rr; // pp指向ii
}
```

这些都合法，但是`rr++`并没有对引用本身做什么增量操作；相反，`++`是应用到了那个`int`上，而这个`int`碰巧就是`ii`。因此，一个引用的值在初始化之后就不可能改变了，它总是引用它的初始化所指称的那个对象。要取得被引用`rr`所引用对象的地址，我们可以写`&rr`。

引用的一种最明显的实现方式是作为一个（常量）指针，在每次使用它时都自动地做间接访问。将引用想像成这种样子不会有任何问题，但要记住的是，一个引用并不是一个对象，不能像指针那样去操作：



在一些情况下，编译器可以通过优化去掉引用，使得在执行时根本不存在任何表示引用的东西。

当引用的初始式是一个左值时（是一个对象，你可以取得它的地址，见4.9.6节），其初始化就是非常简单的事情。对“普通”`T&`的初始式必须是一个类型`T`的左值。

对一个`const T&`的初始式不必是一个左值，甚至可以不是类型`T`的；在这种情况下：

[1] 首先，如果需要将应用到`T`的隐式类型转换（见C.6节）。

[2] 而后将结果存入一个类型*T*的临时变量。

[3] 最后，将此临时变量用做初始式的值。

考虑

```
double& dr = 1;           // 错误：要求左值
const double& cdr = 1;    // ok
```

对后一个初始化的解释是

```
double temp = double(1); // 首先建立一个具有正确值的临时变量
const double& cdr = temp; // 而后用这个临时变量作为cdr的初始式
```

这种保存引用初始式的临时变量将一直存在，直到这个引用的作用域结束。

需要区分对变量的引用和对常量的引用，是因为在变量引用的情况下引进临时量极易出错，对变量的赋值将会变成对于——即将消失的——临时量的赋值。对于常量引用则不会有这类问题，以常量的引用作为函数参数经常是很重要的（11.6节）。

可以通过引用来描述一个函数参数，以使该函数能够改变传递来的变量的值。例如，

```
void increment(int& aa) { aa++; }

void f()
{
    int x = 1;
    increment(x);           // x = 2
}
```

参数传递的语义通过对应的初始化定义，所以，在调用时，*increment*的参数*aa*将变成*x*的另一个名字。为了提高程序的可读性，通常应该尽可能避免让函数去修改它们的参数。相反，你应该让函数明确地返回一个值，或者明确要求一个指针参数：

```
int next(int p) { return p+1; }

void incr(int* p) { (*p)++; }

void g()
{
    int x = 1;
    increment(x);           // x = 2
    x = next(x);            // x = 3
    incr(&x);                // x = 4
}
```

increment(x) 的记法形式不能给读程序的人有关*x*的值可能被修改的提示性信息。而采用*x = next(x)* 和 *incr(&x)* 的形式则可以。因此，如果将“普通”引用参数用于某些函数，那么这些函数的名字就应该给出其引用参数将被修改的强烈提示。

引用还可以用于定义一些函数，使它们既可以被用在赋值的左边，也可以用在右边。同样，许多最有意思的这类应用可以在比较复杂的用户定义类型中找到。作为一个例子，让我们定义一个简单的关联数组。首先，我们定义结构*Pair*如下：

```
struct Pair {
    string name;
    double val;
};
```

基本想法就是让每个`string`有一个关联于它的浮点值。很容易定义一个函数`value()`，让它维护一个`Pair`的数据结构，由曾经提供给它的所有不同的字符串组成。为缩短这个演示，我们在这里采用一个非常简单的（且低效的）实现：

```
vector<Pair> pairs;

double& value(const string& s)
/*
    维护Pair的一个集合；
    检索s，如果找到就返回其值；否则做一个新Pair并返回默认值0
*/
{
    for (int i = 0; i < pairs.size(); i++)
        if (s == pairs[i].name) return pairs[i].val;

    Pair p = { s, 0 };
    pairs.push_back(p); // 将Pair加到最后（3.7.3节）

    return pairs[pairs.size() - 1].val;
}
```

这个功能可以被理解为一个浮点值数组，它以字符串作为下标。对于给定的参数串，`value()`找到对应的浮点对象（而不是对应浮点对象的值），返回到这个对象的一个引用。例如，

```
int main() // 统计每个单词在输入中出现的次数
{
    string buf;
    while (cin >> buf) value(buf)++;

    for (vector<Pair>::const_iterator p = pairs.begin(); p != pairs.end(); ++p)
        cout << p->name << ": " << p->val << '\n';
}
```

每一次`while`循环从标准输入流`cin`将一个单词读进`buf`（3.6节），并更新与它相关联的计数器。最后，结果的表里是输入中遇到的所有互不相同的词，最后将它们及其出现的次数打印出来。举例说，给定输入

```
aa bb bb aa aa bb aa aa
```

程序将产生出

```
aa: 5
bb: 3
```

通过使用模板类，很容易将这个程序进一步精化为一个真正的关联数组类型，带有重载的下标运算符`[]`（11.8节）。利用标准库的`map`（17.4.1节）做这件事就更容易了。

5.6 指向`void`的指针

一个指向任何对象类型的指针都可以赋值给类型为`void*`的变量，`void*`可以赋值给另一个`void*`，两个`void*`可以比较相等与否，而且可以显式地将`void*`转换到另一个类型。其他操作都是不安全的，因为编译器并不知道实际被指的是哪种对象。因此，对`void*`做其他任何操作都将引起编译错误。要使用`void*`，我们就必须显式地将它转换到某个指向特定类型的指针。例如，

```

void f(int* pi)
{
    void* pv = pi; // 可以：从int* 到void* 的隐式转换式
    *pv;           // 错误：void* 不能间接引用
    pv++;          // 错误：void* 不能增量（不知道被指对象的大小）
    int* pi2 = static_cast<int*>(pv); // 显式转换回int*

    double* pd1 = pv; // 错误
    double* pd2 = pi; // 错误
    double* pd3 = static_cast<double*>(pv); // 不安全
}

```

一般来说，如果一个指针被转换（“强制”，*cast*）到与被指对象的实际类型不同的指针，使用一个指针就是不安全的^①。例如，一台机器可能假设每个*double*被分配了8个字节，如果真是这样的话，那么，若被*pi*所指的*int*不是这样分配就会产生奇怪的行为^②。这种形式的显式类型转换，从本质上说就是不安全的、丑陋的，所以这里所用的记法*static_cast*，也被设计成极其丑陋的样子。

*void** 的最重要用途是需要向函数传递一个指针，而又不能对对象的类型做任何假设。还有就是从函数返回一个无类型的对象。要使用这样的对象，必须通过显式的类型转换。

采用*void** 的函数通常存在于系统中很低的层次里，在那里需要操作某些真实的硬件资源。例如，

```
void* my_alloc(size_t n); // 从我特定的堆中分配n字节的存储
```

在系统中较高层次上出现*void** 应该认为是可疑的，它们就像是设计错误的指示器。在那些为优化而使用的地方，可以将*void** 隐藏在类型安全的界面之后（13.5节、24.4.2节）。

到函数的指针（7.7节）和到成员的指针（15.5节）都不能赋给*void**。

5.7 结构

数组是相同类型的元素的一个聚集。一个*struct*则是（差不多）任意类型元素的一个聚集。例如，

```

struct address {
    char* name;           // "Jim Dandy"
    long int number;      // 61
    char* street;         // "South St"
    char* town;           // "New Providence"
    char state[2];        // 'N' 'J'
    long zip;             // 7974
};

```

这定义了一个新类型，名为*address*，它由一些你所需要的以便能给某人发送邮件的条目组成。请注意最后的分号。这是C++里很少有的几处在花括号之后还需要写分号的地方，因此人们很容易忘记它。

① 原话不够清楚确切，这里做了修正。对于上面的例子，*pv*所指的是一个整数对象（由*pi*得到）。最后一个语句将这个指针转换后赋给*double** 变量*pd3*。通过*pd3*间接访问是不安全的。——译者注

② 这一解释还不够。因为，即使*int*也是8个字节，它所采用的二进制表示模式也必定与*double*不同，这样，对*pd3*的间接访问仍然会产生奇怪的行为。——译者注

类型为`address`的变量可以像其他变量一样声明，其中的各个成员可以通过`.`（圆点）运算符访问。例如，

```
void f()
{
    address jd;
    jd.name = "Jim Dandy";
    jd.number = 61;
}
```

用于对数组初始化的记法形式也适用于初始化结构类型的变量。例如，

```
address jd = {
    "Jim Dandy",
    61, "South St",
    "New Providence", {'N', 'J'}, 7974
};
```

当然，一般来说采用构造函数（10.2.3节）更好一些。请注意，`jd.state`不能用字符串`"NJ"`去初始化。字符串总以字符`'\0'`结束，这样，`"NJ"`实际上就有三个字符——比`jd.state`的需要多了一个。

结构对象常常通过指针用`->`运算符（结构指针间接）访问。例如，

```
void print_addr(address* p)
{
    cout << p->name << '\n'
         << p->number << ' ' << p->street << '\n'
         << p->town << '\n'
         << p->state[0] << p->state[1] << ' ' << p->zip << '\n';
}
```

当`p`是指针时，`p->m`等价于`(*p).m`。

结构类型的对象可以被赋值、作为函数参数传递、作为函数的返回值返回。例如，

```
address current;

address set_current(address next)
{
    address prev = current;
    current = next;
    return prev;
}
```

其他可能的运算符，例如比较（`==` 和 `!=`）都没有定义。但是用户可以自己定义这些运算符（第11章）。

结构类型对象的大小未必是其成员的大小之和，这是因为许多机器要求将确定类型的对象分配在某种与机器的系统结构有关的边界上，或者是在采用适当分配的情况下能更有效地处理这些对象。例如，整数常常被分配在机器字的边界上。在这类机器上，对象被称为具有对齐的性质。对齐会在结构中造成“空洞”。例如，在许多机器上，`sizeof(address)`是24，而不是人们可能期望的22。你可以通过简单地从大到小排列成员，以取得最小的空间浪费（最大的成员在先）。但是，最好还是按照可读性的要求去排列它们，只在那些必须优化的地方按大小将它们排好顺序。

类型的名字在出现之后立即就可以使用了，不必等到看到完整的声明之后。例如，

```
struct Link {
    Link* previous;
    Link* successor;
};
```

在完整声明被看到之前，不能去声明这个结构类型的新对象。例如，

```
struct No_good {
    No_good member;    // 错误：递归定义
};
```

这是一个错误，因为编译将无法确定`No_good`的大小。要想允许两个（或更多）结构类型互相引用，我们可以先将一个名字声明为结构的名称。例如，

```
struct List;    // 后面定义

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
};

struct List {
    Link* head;
};
```

如果没有`List`的第一个声明，在`Link`的声明里使用`List`就会导致一个语法错误。

结构类型的名字可以在这个类型的定义之前使用，只要在有关使用中无需知道成员的名字或者结构的大小。例如，

```
class S;    // S是某个类型的名字

extern S a;
S f();
void g(S);
S* h(S*);
```

当然，除非类型`S`已经定义，否则许多其他声明都是无法做的：

```
void k(S* p)
{
    S a;        // 错误：S无定义；分配时需要其大小
    f();        // 错误：S无定义；返回值需要其大小
    g(a);       // 错误：S无定义；传递参数需要其大小
    p->m = 7;    // 错误：S无定义；成员名未知

    S* q = h(p); // 可以：指针可以分配和传递
    q->m = 7;    // 错误：S无定义；成员名未知
}
```

`struct`是`class`（第10章）的简单形式。

由于与C的早期历史接轨的原因，在同一个作用域里，允许一个名字被同时用于一个`struct`和一个非结构实体。例如，

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

在这种情况下，简单的名字本身（`stat`）就是那个非结构物的名字，而该结构则必须通过加前

缀`struct`的方式引用。类似地，关键字`class`、`union`（C.8.2节）和`enum`（4.8节）也可用做前缀以消除歧义。当然，最好是不要去重载某个名字，不要造成这种必要性。

5.7.1 类型等价

两个结构总是不同的类型，即使它们有着相同的成员。例如，

```
struct S1 { int a; };
struct S2 { int a; };
```

是不同的类型。所以

```
S1 x;
S2 y = x; // 错误：类型不匹配
```

结构类型也与各种基本类型不同，所以

```
S1 x;
int i = x; // 错误：类型不匹配
```

每个`Struct`都必须是在程序里惟一定义的（9.2.3节）。

5.8 忠告

- [1] 避免非平凡的指针算术；5.3节。
- [2] 当心，不要超出数组的界线去写；5.3.1节。
- [3] 尽量使用0而不是`NULL`；5.1.1节。
- [4] 尽量使用`vector`和`valarray`而不是内部（C风格）的数组；5.3.1节。
- [5] 尽量使用`string`而不是以0结尾的`char`数组；5.3节。
- [6] 尽量少用普通的引用参数；5.5节。
- [7] 避免`void*`，除了在某些低级代码里；5.6节。
- [8] 避免在代码中使用非平凡的文字量（“神秘的数”）。相反，应该定义和使用各种符号常量。4.8节、5.4节。

5.9 练习

1. (*1) 写出下面声明：一个到字符的指针；一个包含10个整数的数组；一个到包含10个整数的数组的引用；一个到字符串的数组的指针；一个到字符的指针的指针；一个常量整数；一个到常量整数的指针；一个到整数的常量指针。并为每个声明做初始化。
2. (*1.5) 在你的系统上，对于指针类型`char*`、`int*`、`void*`有什么限制吗？例如，`int*`可以具有奇数的值吗？提示：对齐问题。
3. (*1) 用`typedef`去定义类型`unsigned char`，`const unsigned char`，到整数的指针，到字符的指针的指针，到字符的数组的指针，7个到整数的指针的数组，到包含7个到整数的指针的数组的指针，包含8个数组的数组，其中每个数组包含7个到整数的指针。
4. (*1) 写一个函数，它交换两个整数（交换它们的值）。用`int*`作为参数类型。再写另一个交换函数，用`int&`作为参数类型。
5. (*1.5) 在下面例子里，数组`str`的大小是什么？

```
char str[] = "a short string";
```

"a short string" 的长度是多少?

6. (*1) 定义了函数 `f(char)`, `g(char&)`, `h(const char&)` 之后, 用参数 'a', 49, 3300, `c`, `uc`, `sc` 作为参数调用它们, 其中 `c` 是 `char`, `uc` 是 `unsigned char`, `sc` 是 `signed char`。哪些调用是合法的? 哪些调用将导致编译器引进临时变量?
7. (*1.5) 定义一个包含一年中各个月份的名字和每个月的天数的表格。输出这个表。做这件事情两次: 第一次用一个 `char` 的数组表示名字, 用另一个数组表示天数; 另一次用一个结构的数组, 在每个结构中保存一个月的名字和它的天数。
8. (*2) 运行一些测试, 看看你的编译器对于用指针写出的迭代和用下标写出的迭代 (5.3.1节) 是否真的生成相同的代码。如果能要求不同级别的优化, 请看看这些优化将如何影响所生成的代码的质量。
9. (*1.5) 找一个例子, 其中将一个名字用于它自己的初始式是有意义的。
10. (*1) 定义一个字符串的数组, 其中的字符串保存的是月份的名字。打印出这些字符串。将这些字符串传递给一个函数去打印出这些字符串。
11. (*2) 从输入读一系列的单词, 使用 `Quit` 作为输入的结束单词。按照读入的顺序打印出这些单词, 但同一个单词不要打印两次。修改这个程序, 在打印之前对单词排序。
12. (*2) 写一个函数, 它统计在一个 `string` 里一对字母出现的次数。写另一个函数对以零结束的 `char` 的数组 (C 风格的字符串) 做同样事情。例如, 字符对 "ab" 在 "xabaacbaxabb" 里出现了两次。
13. (*1.5) 定义一个 `struct Date` 以保存日期的轨迹。提供一些函数, 从输入读 `Date`, 向输出写 `Date`, 以及用一个日期去初始化 `Date`。

第6章 表达式和语句

不成熟的优化是万恶之源。

——D. Knuth

另一方面，

我们无法对效率问题视而不见。

——Jon Bentley

桌面计算器实例——输入——命令行参数——表达式一览——逻辑和关系运算符——
增量和减量——自由存储——显式类型转换——语句一览——声明——选择语句——
条件里的声明——迭代语句——声名狼藉的`goto`——注释和缩进编排——忠告——
练习

6.1 一个桌面计算器

这里要介绍各种语句和表达式，将通过一个桌面计算器的程序做这件事情，该计算器提供了四种作为浮点数的中缀运算符的标准算术运算。例如，给了输入

```
r = 2.5  
area = pi * r * r
```

(`pi`预先有定义)，计算器程序将写出

```
2.5  
19.635
```

这里的2.5是第一行输入的结果，19.635是第二行的结果。

这个计算器由四个部分组成：一个分析器，一个输入函数，一个符号表和一个驱动程序。实际上可以将它看成一个最小的编译器，其中的分析器做语法分析工作，输入函数处理输入和词法分析，符号表保存持久性信息，驱动程序处理初始化、输出和错误等。我们还可以给这个计算器加入许多功能，以使其更加有用（6.6[20]）。但是像它目前的这个样子，代码已经很长了。加进更多新功能并不会提供对C++使用的更多见识。

6.1.1 分析器

这里是该计算器所接受的语言的语法：

```
program:  
    END // END表示输入结束  
    expr_list END
```

```

expr_list:
    expression PRINT          // PRINT是分号
    expression PRINT expr_list.

expression:
    expression + term
    expression - term
    term

term:
    term / primary
    term * primary
    primary

primary:
    NUMBER
    NAME
    NAME = expression
    - primary
    ( expression )

```

换句话说，一个程序是由分号分隔的一系列表达式。表达式的基本单元是数、名字和运算符*、/、+、-（包括一元的和二元的）和=。名字在使用之前无需声明。

语法分析采用通常称为递归下降的风格，这是一种很流行的直截了当的自顶向下技术。在像C++这类语言中，函数调用的代价相对较低，采用这种技术就相当有效。对于语法中的每个产生式存在着一个函数，它还要调用其他的函数。终结符（例如**END**、**NUMBER**、+和-）由词法分析程序**get_token()**识别，而非终结符则由语法分析函数**expr()**、**term()**和**prim()**识别。一旦一个（子）表达式的两个运算对象都已经知道了，就立即对这个表达式求值。而对于真正的编译器，在这一点则可能是生成代码。

分析器利用函数**get_token()**取得输入，最后一次调用**get_token()**得到的值可以在全局变量**curr_tok**里找到。**curr_tok**的类型是枚举**Token_value**：

```

enum Token_value {
    NAME,          NUMBER,      END,
    PLUS='+',      MINUS='-',    MUL='*',          DIV='/',
    PRINT=';',      ASSIGN='=',  LP='(',          RP=')'
};

Token_value curr_tok = PRINT;

```

将单词（token）用它们的字符所对应的整数表示，这样做既方便有效，又能帮助使用排错系统的人。只要在所有输入字符中，没有任何字符的值被当做枚举符使用，这种方式就可以工作——在我所知道的字符集中，都不存在可打印字符的值只有一位数字的情况。我选择**PRINT**作为**curr_tok**的初始值，因为这正是计算器计算完一个表达式，并显示结果值之后**curr_tok**应该具有的值。这样，我也就是以一种正常状态“启动这个系统”，这样做最大限度地减少了出错的机会和对特殊启动代码的需求。

每个分析函数都有一个**bool**（4.2节）参数，指明该函数是否需要调用**get_token()**去取得下一个单词。每个分析函数都将对“它的”表达式求值并返回这个值。函数**expr()**处理加和减。它由一个查找被加减的项的循环组成：

```

double expr(bool get)          // 加和减
{

```

```

double left = term(true);
for (;;) // “永远”
    switch (curr_tok) {
        case PLUS:
            left += term(true);
            break;
        case MINUS:
            left -= term(true);
            break;
        default:
            return left;
    }
}

```

这个函数本身并不做多少事情，它以一个小程序里高层函数的典型方式，调用其他函数去完成具体工作。

*switch-statement*对照着一集常量，检查在关键字*switch*之后的括号里所描述的条件值。*break*语句用于从开关语句中退出。在各个*case*之后的常量必须互不相同。如果被检查的值与任何*case*标号都不匹配，那么就选择*default*。程序员不一定要提供*default*。

请注意，形如 $2 - 3 + 4$ 这样的表达式将被按 $(2 - 3) + 4$ 的方式求值，正也是语法所描述的。

奇特的记法形式*for*(;;)是人们描述无限循环的一种标准方式，你可以将它读作“永远”。这是*for-statement* (6.3.3节)的一种退化形式；我们也可以用*while(true)*代替它。这里的开关语句将反复执行，直到遇到的不是+和-，这就会导致在*default*情况中的返回语句的执行。

运算符+=和-=分别用于处理加和减；可以用`left = left + term(true)`和`left = left - term(true)`代替它们而不会影响程序的意义。然而，`left += term(true)`和`left -= term(true)`不仅更短一些，而且也更直接地描述了我们想做的事情。每个赋值运算符是一个单独的词法单词，因此`a += 1;`是个语法错误，因为在+和=之间出现了空格。

下面这些二元运算符都有相应的赋值运算符：

+ - * / % & | ^ << >>

所以下面赋值运算符都可以使用

= += -= *= /= %= &= |= ^= <<= >>=

这里的%是取模，或说是求余运算符；&、|和^分别是按位逻辑“与”、按位逻辑“或”和按位逻辑“异或”运算符；<<和>>是左移和右移运算符。6.2节总结了所有运算符和它们的意义。对于一个应用于内部类型运算对象的二元运算符@，表达式`x @= y`的意思就是`x = x @ y`，不过在前一种情况中只对x求值一次。

第8章和第9章将讨论如何将程序组织为一组模块。除了一个例外之处，这个计算器例子中的所有声明都可以排列好，使每个东西都只需要在它被使用之前声明一次。这个例外就是*expr()*，它调用*term()*，这个函数转而调用*prim()*，而*prim()*又调用了*expr()*。这种循环调用的情况必须打破，只要将一个声明

```
double expr(bool);
```

放在*prim()*的定义之前就行了。

函数*term()*处理乘和除，采用的方式与*expr()*处理加和减的方式一样：

```

double term(bool get)          // 乘和除
{
    double left = prim(get);
    for (;;)
        switch (curr_tok) {
            case MUL:
                left *= prim(true);
                break;
            case DIV:
                if (double d = prim(true)) {
                    left /= d;
                    break;
                }
                return error("divide by 0");
            default:
                return left;
        }
}

```

用0去除的结果无定义，通常将是个灾难。因此我们需要在除之前检查是否为0，如果检查到以0作为除数就调用`error()`。函数`error()`将在6.1.4节讨论。

变量`d`正是在需要它的地方才被引进程序里，而且立即做了初始化。一个在条件中引进的名字，其作用域就是这个条件所控制的语句，其结果值被作为这个条件的值（6.3.2.1节）。这样，只有在`d`不是0的情况下除法和赋值`left /= d`才会进行。

函数`prim()`处理初等项的方式很像`expr()`和`term()`，当然，因为我们需要进入调用层次的更下一层，在这里需要多做一点事情，而且不必做循环：

```

double number_value;
string string_value;

double prim(bool get)          // 处理初等项
{
    if (get) get_token();

    switch (curr_tok) {
        case NUMBER:           // 浮点常量
            { double v = number_value;
              get_token();
              return v;
            }
        case NAME:
            { double& v = table[string_value];
              if (get_token() == ASSIGN) v = expr(true);
              return v;
            }
        case MINUS:             // 一元
            return -prim(true);
        case LP:
            { double e = expr(true);
              if (curr_tok != RP) return error("(" expected");
              get_token();       // 吃掉 ')'
              return e;
            }
        default:
            return error("primary expected");
    }
}

```


如果遇到一个**NUMBER**（也就是说，遇到一个整数或浮点的文字量），*prim()*就返回它的值。输入例程*get_token()*将把有关的值存入全局变量*number_value*。在程序里采用全局变量，一般说明这种结构不是很清晰——应该进行某种优化。目前就出现了这种情况。理想方式是让一个词法单词由两个部分组成：一个刻画单词种类（在这个程序里是*Token_value*）的值和（如果需要的话）一个单词值。在这里只存在一个单独的变量*curr_tok*，所以需要用一个全局变量*number_value*保存最近读到的一个**NUMBER**值。清除这个不良的全局变量的工作留做练习（6.6[21]）^②。在调用*get_token()*之前，我们并不需要将*number_value*的值保存到某个局部变量*v*里，因为对于合法的输入，该计算器在去读其他形式的输入之前只需要用一个数。当然，将这种值存起来并在出错时显示，也能对用户有所帮助。

按照将最近的一个**NUMBER**的值保存于*number_value*的同样方式，代表最近遇到的**NAME**的字符串将被保存在*string_value*里。在对一个名字做其他事情之前，计算器必须首先去看是要对它赋值，还是要去读它的值。这两种情况都需要检查符号表，该符号表是一个*map*（3.7.4节、17.4.1节）：

```
map<string, double> table;
```

也就是说，如果用一个*string*去索引*table*，得到的结果是对应于该*string*的*double*值。例如，如果用户输入

```
radius = 6378.388;
```

计算器将执行

```
double& v = table["radius"];
// ... expr() 计算要赋的值 ...
v = 6378.388;
```

在*expr()*由输入字符序列计算6378.388的值的過程中，将通过引用*v*去把握住与*radius*相关联的那个*double*值。

6.1.2 输入函数

读输入的部分通常都是程序里最麻烦的部分，这是因为程序必须与人交流，它必须处理人们的奇思怪想、各种习惯和看起来像是随机的错误。试图强求人按照某种更适合机器的方式活动通常被（正确地）认为是很令人讨厌的。低级输入例程的工作是读入字符，并由它们组合出高级的单词，这些单词随之成为高级例程的输入单位。在这里的低级输入由*get_token()*完成。写低级输入例程并不是每天都要做的工作，许多系统中提供了完成这些事情的标准函数。

我将分两个阶段完成*get_token()*。首先我将提供一个不太靠得住的简单版本，它给用户强加了很大负担。而后，我要把它修改成一个稍微不那么优雅，但却更容易使用的版本。

基本想法是去读一个字符，根据它决定需要去组合的是哪种单词，而后返回表示被读单词的*Token_value*值。

初始化语句将第一个非空白字符读入*ch*，并检查该读入操作的成功完成：

```
Token_value get_token()
{
```

② 实际上，这个程序里的全局变量远不止*number_value*一个。请读者仔细检查程序，看看总共有哪些全局变量，也想一想如何消除它们，或者消除它们会遇到的问题。——译者注

```

char ch = 0;
cin >> ch;

switch (ch) {
case 0:
    return curr_tok=END;    // 赋值和返回

```

按默认方式，运算符 `>>` 将跳过空白（即空格、制表符、换行符等），如果操作失败还将保持 `ch` 不变。因此，`ch == 0` 的情况就表示了输入的结束。

赋值也是一个运算符，赋值表达式的结果就是赋给变量的那个值。这使我能在同一个语句里将值 `END` 赋给 `curr_tok`，并且还返回它。在一个语句中做两件事对于维护很有帮助，如果赋值和返回在代码中分开做，程序员就可能会修改了一个而忘记了另一个。

在考虑整个函数之前，让我们先分开来看一些情况。表达式结束符 `';`，括号和运算符的处理都很简单，只需要返回它们的值：

```

case ';' :
case '*' :
case '/' :
case '+' :
case '-' :
case '(' :
case ')' :
case '=' :
    return curr_tok=Token_value(ch);

```

数可以按如下方式处理：

```

case '0' : case '1' : case '2' : case '3' : case '4' :
case '5' : case '6' : case '7' : case '8' : case '9' :
case '.' :
    cin.putback(ch);
    cin >> number_value;
    return curr_tok=NUMBER;

```

按水平方式将 `case` 标号堆在一起而不是垂直地放置并不是什么好主意，因为这种安排更难阅读。然而，在每行中放一个数字也很令人生厌。由于运算符 `>>` 已经有针对将浮点常数读入 `double` 的定义，这里的代码就非常简单：首先把第一个字符（数字或者圆点）放回 `cin`，而后将常数读到 `number_value` 里。

名字的处理也与此类似：

```

default:                // NAME, NAME =, 或者错误
    if (isalpha(ch)) {
        cin.putback(ch);
        cin >> string_value;
        return curr_tok=NAME;
    }
    error("bad token");
    return curr_tok=PRINT;

```

这里用了标准库函数 `isalpha()`（20.4.2节），以避免将每个字母列为一个 `case` 标号。运算符 `>>` 用于字符串（这里是 `string_value`）的结果是不断读入字符，直到遇到空白为止。这样，用户要想把某个名字作为运算符的运算对象，就必须用空白表示名字的结束。这一情况当然不太理想，我们将在6.1.3节重新回到这个问题。

这里是最后的完整的输入函数：

```
Token_value get_token()
{
    char ch = 0;
    cin >> ch;
    switch (ch) {
        case 0:
            return curr_tok=END;

        case ' ':
        case '*':
        case '/':
        case '+':
        case '-':
        case '(':
        case ')':
        case '=':
            return curr_tok=Token_value(ch);

        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        case '.':
            cin.putback(ch);
            cin >> number_value;
            return curr_tok=NUMBER;

        default: // NAME, NAME =, 或者错误
            if (isalpha(ch)) {
                cin.putback(ch);
                cin >> string_value;
                return curr_tok=NAME;
            }
            error("bad token");
            return curr_tok=PRINT;
    }
}
```

由于前面将运算符的`Token_value`值定义成该运算符的整数值（4.8节），从运算符到对应的单词值的转换就非常简单了。

6.1.3 低级输入

使用至今所定义的这个计算器，可以发现一些不方便的地方：需要记住在每个表达式的最后加一个分号，以使它的值能够被打印出来；每个名字都需要用空白结束等。这些都非常令人讨厌。例如，`x=7`将被作为一个标识符，而不是标识符`x`后跟运算符`=`和数`7`。用读入单个字符的代码来代替`get_token()`里那种基于类型的默认输入操作，就可以解决这两个问题。

首先，我们可将换行符号看做与分号等价，也当做表达式结束：

```
Token_value get_token()
{
    char ch;

    do { // 跳过空白，除了 '\n'
        if (!cin.get(ch)) return curr_tok = END;
    } while (ch != '\n' && isspace(ch));
```

```

switch (ch) {
case ' ':
case '\n':
    return curr_tok=PRINT;

```

这里用了一个`do`语句，它等价于`while`语句，除了被控制的语句至少总要执行一次之外。调用`cin.get(ch)`从标准输入流读一个字符到`ch`里。按照默认方式，`get`不会像`>>`运算符那样跳过空白。检测`if(!cin.get(ch))`在无法从`cin`读到字符时为真；在这种情况下，就返回`END`以结束计算器的本次执行。这里用了运算符`!`（否定），因为`get()`在成功的情况下将返回`true`。

标准库函数`isspace()`提供了对空白的标准检测（20.4.2节）。当`c`是空白字符时，`isspace(c)`将返回非0值，否则就返回0。这个测试是通过查表的方式实现的，因此，使用`isspace()`比直接检测各个空白字符要快得多。类似的函数包括检测一个字符是否是数字——`isdigit()`，是否是字母——`isalpha()`，是否是数字或字母——`isalnum()`。

在跳过空白之后，下一个字符将用于确定读来的是哪类词法单词。

采用`>>`读入字符串直到遇到空白会引起问题，这一问题可以通过一次读一个字符，直到遇到非字母非数字字符的方式解决：

```

default:                // NAME, NAME=, 或者错误
    if (isalpha(ch)) {
        string_value = ch;
        while (cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
        cin.putback(ch);
        return curr_tok=NAME;
    }
    error("bad token");
    return curr_tok=PRINT;

```

幸运的是，这两个改进的实现都只要修改一小段局部代码。构造出这样的程序，使程序的改进能通过局部性修改实现，这也是一个非常重要的设计目标。

6.1.4 错误处理

由于这个程序如此简单，错误处理也就没有作为一个主要的考虑。这里的错误处理程序简单地统计错误次数，写出一条错误信息后返回：

```

int no_of_errors;

double error(const string& s)
{
    no_of_errors++;
    cerr << "error: " << s << '\n';
    return 1;
}

```

流`cerr`是一个非缓冲的输出流，通常用于报告错误（21.2.1节）。

在这里返回一个值，这是因为错误通常是在表达式求值期间遇到的，因此我们就需要或者是让这次计算整个流产，或者是返回一个不大会在后面引起问题的值。对这个简单计算器，采用后一种方式是很合适的。让`get_token()`记录行的编号，将使`error()`能够通知用户出错的大致位置，这一做法对于非交互式地使用计算器将很有帮助（6.6[9]）。

常有的情况是，程序在出现了一个错误后就必须终止，因为我们无法安排一种有意义的

继续方式。需要这样做时可以调用`exit()`，该函数将首先清理诸如输出流等，而后结束这一程序，并以自己的参数作为程序的返回值（9.4.1.1节）。

更有特色的错误处理机制可以通过异常（8.3节、第14章）实现，但对于一个大约150行的小计算器而言，我们现在的做法已经很合适了。

6.1.5 驱动程序

有了所有这些程序片段之后，我们还需要一个驱动程序，以便使所有的事情能够开始。对这个简单的例子，`main()`就可以完成这件事：

```
int main()
{
    table["pi"] = 3.1415926535897932385; // 插入预定义的名字
    table["e"] = 2.7182818284590452354;

    while (cin) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    return no_of_errors;
}
```

按照惯例，`main()`在正常结束时应该返回0，否则就返回非0（3.2节）。返回出现的错误数正好与此相符。在这里，需要做的初始化工作只有将一些预定义的名字插入符号表中。

主循环的基本工作就是读入表达式并写出回答，这可以由下面一行完成：

```
cout << expr(false) << '\n';
```

参数`false`告诉`expr()`它无须调用`get_token()`去取得要用的单词。

每轮循环都需要检测`cin`，这样就能保证在输入流出了什么问题时使程序结束。检测`END`则保证在`get_token()`遇到文件结束时能正确地退出循环。一个`break`语句将导致退出最内层的`switch`语句或者循环（即，`for`语句、`while`语句或者`do`语句）。检测`PRINT`（即`'\n'`或者`','`）将使`expr()`不必去处理空的表达式。`continue`语句等价于跳到循环的最后，所以在这里的

```
while (cin) {
    // ...
    if (curr_tok == PRINT) continue;
    cout << expr(false) << '\n';
}
```

等价于

```
while (cin) {
    // ...
    if (curr_tok != PRINT)
        cout << expr(false) << '\n';
}
```

6.1.6 头文件

这个计算器使用了一些标准库功能，因此必须把适当的头文件通过`#include`包含到完整

的程序里：

```
#include<iostream> // I/O
#include<string>    // 字符串
#include<map>       // 映射
#include<cctype>    // isalpha() 等
```

所有这些头文件提供的功能都在`std`名字空间里，所以，要使用头文件中所提供的名字，我们或者是需要显式地加上限定词`std::`，或者要通过下面的行，将这些名字都放入全局名字空间

```
using namespace std;
```

为了避免将有关表达式的讨论与有关模块化的讨论相混淆，我采取了后一种做法。第8章和第9章将讨论用名字空间把这个计算器组织到模块里的一些方式，以及如何将它组织在一些源文件里的有关问题。在许多系统上，标准头文件都有带`.h`后缀的等价文件，它们声明了同样的类、函数等，并把所有这些都放进全局名字空间（9.2.1节、9.2.4节、B.3.1节）。

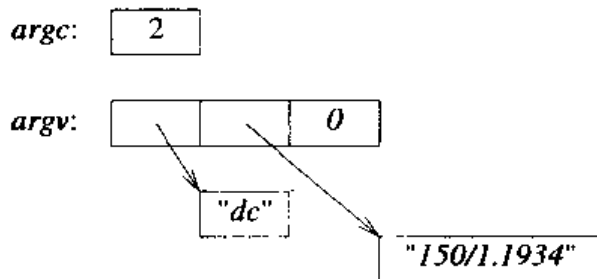
6.1.7 命令行参数

在写出这个程序和测试过之后，我发现首先启动程序，而后键入表达式，最后结束，这些也是很麻烦的事情。我最常用的就是对一个表达式求值。如果能让这个表达式表现为命令行的参数，就可以避免一些键入。

一个程序开始于对函数`main()`的调用（3.2节、9.4节）。在这样做的时候，有两个参数被送给`main()`，其中的一个描述了命令行参数的个数，通常称为`argc`；另一个是命令行参数的数组，通常称为`argv`。命令行参数都是字符串，所以`argv`的类型是`char* [argc + 1]`。该程序的名字（如它在命令行里出现的那样）也作为`argv[0]`传进来，所以`argc`至少是1。这个参数的表总以0结束，也就是说，`argv[argc] == 0`。例如，对于命令

```
dc 150/1.1934
```

命令行参数具有如下的值：



由于调用`main()`的规定是与C共享的，所以这里用的是C风格的数组和字符串。

取得命令行参数并不难，问题是怎样通过最少的重新编程去使用它。这里的想法是按照我们从输入流读入的同样方式从命令行读入。从字符串读入的流被称为`istream`，这一点也不奇怪。不幸的是，并不存在某种优雅的方式使`cin`转去引用一个`istream`。所以我们必须找到一种方式，让计算器的输入函数去引用一个`istream`。进一步说，我们必须找到一种方式，使计算器程序能够根据所提供的命令行的情况，确定是去引用一个`istream`，还是引用`cin`。

一种简单的方法是引进一个全局的指针`input`，让它指向要使用的输入流，并让每个输入

例程都使用它：

```
istream* input; // 指向输入流

int main(int argc, char* argv[])
{
    switch (argc) {
        case 1: // 从标准输入读
            input = &cin;
            break;
        case 2: // 从参数字符串读
            input = new istringstream(argv[1]);
            break;
        default:
            error("too many arguments");
            return 1;
    }

    table["pi"] = 3.1415926535897932385; // 插入预定义的名字
    table["e"] = 2.7182818284590452354;

    while (*input) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    if (input != &cin) delete input;
    return no_of_errors;
}
```

一个`istringstream`也是一类`istream`，它从自己的字符串参数读入（21.5.3节）。在遇到自己的字符串的结束时，`istringstream`将以其他流遇到输入结束的同样方式失败（3.6节、21.3.3节）。如果要使用`istringstream`，你就必须包含`<sstream>`。

修改`main()`，使之能接受几个命令行参数也不难，但看来并没有这种必要性。特别是几个表达式完全可以作为一个参数传递：

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

我在这里使用了引号，这是因为“;”在我的UNIX系统里是命令分隔符。其他系统对于在启动时程序所提供参数也可能有不同的规定。

要修改所有的输入例程，让它们都使用`*input`而不是`cin`，以便取得替换输入源的灵活性，这种做法并不优雅。如果我早有先见之明，在一开始就引进某种类似`input`的东西，实际上就可以避免这种改变。存在着一种更一般且有用的观点：应注意到输入源实际上应该作为这个计算器模块的参数。这也就是说，有关这个计算器实例的基本问题是，我说的所谓“计算器”不过是一组函数和数据。并不存在明确表示计算器的模块（2.4节）或者对象（2.5.2节）。如果我原来提出的问题是设计一个计算器模块或者计算器类型，我可能早就很自然地会去考虑以什么作为它的参数了（8.5[3]、10.6[16]）。

6.1.8 有关风格的注记

对于不熟悉关联数组的程序员而言，采用标准库的`map`作为符号表似乎是一种作弊。这当

然不是。标准库和其他库都是为了使用。一般来说，对于一个库的设计和实现所付出的努力，远远多于一个程序员为他手工做出的只为在一个程序里使用的代码片段。

请看计算器的代码，特别是第一个版本，我们可以看到这里并没有出现多少C风格的、低级的代码。许多传统的技巧细节都被用标准库类例如`ostream`、`string`和`map`（3.4节、3.5节、3.7.4节、第17章）等取代了。

请注意，这里算术、循环以及赋值都比较少。这就是在那些并不直接操作硬件或者实现低级抽象的代码所应有的形式。

6.2 运算符概览

本节将给出对表达式的综述和一些实例。每个运算符都伴随着一个或几个经常对它使用的名字和一个使用的例子。在表格中，`class_name`表示一个类的名字，`member`表示一个成员的名字，`object`表示一个能产生出类对象的表达式，`pointer`是一个产生指针的表达式，`expr`是表达式，而`lvalue`是一个表示非常量对象的表达式。其中的`type`如果出现在括号里，它就可以是一个一般性的类型名（可以带`*`，`()`等），否则对`type`的使用就存在着某些限制（A.5节）。

表达式的语法与运算对象的类型无关。这里所给出的意义只适用于内部类型的运算对象（4.1.1节）。除此之外，你可以定义运算符的意义，使之能应用于用户定义类型的运算对象（2.5.2节、第11章）。

运算符概览	
作用域解析	<code>class_name::member</code>
作用域解析	<code>namespace_name::member</code>
全局	<code>::name</code>
全局	<code>::qualified-name</code>
成员选择	<code>object.member</code>
成员选择	<code>pointer->member</code>
下标	<code>pointer[expr]</code>
函数调用	<code>expr(expr_list)</code>
值构造	<code>type(expr_list)</code>
后增量	<code>lvalue++</code>
后减量	<code>lvalue--</code>
类型识别	<code>typeid(type)</code>
运行时类型识别	<code>typeid(expr)</code>
运行时检查的转换	<code>dynamic_cast<type>(expr)</code>
编译时检查的转换	<code>static_cast<type>(expr)</code>
不检查的转换	<code>reinterpret_cast<type>(expr)</code>
<code>const</code> 转换	<code>const_cast<type>(expr)</code>
对象的大小	<code>sizeof expr</code>
类型的大小	<code>sizeof(type)</code>
前增量	<code>++ lvalue</code>
前减量	<code>-- lvalue</code>
补	<code>~ expr</code>
非（否定）	<code>! expr</code>
一元负号	<code>- expr</code>

运算符概览 (续)	
一元止号	<i>+ expr</i>
地址	<i>& lvalue</i>
间接	<i>* expr</i>
建立 (分配)	<i>new type</i>
建立 (分配并初始化)	<i>new type (expr-list)</i>
建立 (放置)	<i>new (expr-list) type</i>
建立 (放置并初始化)	<i>new (expr-list) type (expr-list)</i>
销毁 (释放)	<i>delete pointer</i>
销毁数组	<i>delete [] pointer</i>
强制 (类型转换)	<i>(type) expr</i>
成员选择	<i>object . * pointer-to-member</i>
成员选择	<i>pointer -> * pointer-to-member</i>
乘	<i>expr * expr</i>
除	<i>expr / expr</i>
取模 (余数)	<i>expr % expr</i>
加 (求和)	<i>expr + expr</i>
减 (求差)	<i>expr - expr</i>
左移	<i>expr << expr</i>
右移	<i>expr >> expr</i>
小于	<i>expr < expr</i>
小于等于	<i>expr <= expr</i>
大于	<i>expr > expr</i>
大于等于	<i>expr >= expr</i>
等于	<i>expr == expr</i>
不等于	<i>expr != expr</i>
按位与	<i>expr & expr</i>
按位异或	<i>expr ^ expr</i>
按位或	<i>expr expr</i>
逻辑与	<i>expr && expr</i>
逻辑或	<i>expr expr</i>
条件表达式	<i>expr ? expr : expr</i>
简单赋值	<i>lvalue = expr</i>
乘并赋值	<i>lvalue *= expr</i>
除并赋值	<i>lvalue /= expr</i>
取模并赋值	<i>lvalue %= expr</i>
加并赋值	<i>lvalue += expr</i>
减并赋值	<i>lvalue -= expr</i>
左移并赋值	<i>lvalue <<= expr</i>
右移并赋值	<i>lvalue >>= expr</i>
与并赋值	<i>lvalue &= expr</i>
或并赋值	<i>lvalue = expr</i>
异或并赋值	<i>lvalue ^= expr</i>
抛出异常	<i>throw expr</i>
逗号 (序列)	<i>expr, expr</i>

每个间隔里的运算符具有相同优先级。位于上面的间隔里的运算符比下面间隔里的运算符优

优先级更高。例如, $a + b * c$ 的意思是 $a + (b * c)$, 而不是 $(a + b) * c$, 因为 $*$ 比 $+$ 的优先级更高。类似地, $*p++$ 的意思是 $*(p++)$ 而不是 $(*p)++$ 。

一元运算符和赋值运算符是右结合的, 其他运算符都是左结合的。例如, $a = b = c$ 的意思是 $a = (b = c)$, $a + b + c$ 是 $(a + b) + c$ 。

也有不多的几条语法规则无法通过优先级 (也被看做是约束强度) 和结合性来说明。例如, $a = b < c ? d = e : f = g$ 的意思是 $a = ((b < c) ? (d = e) : (f = g))$ 。你需要去查看语法 (A.5 节) 以确定这些东西。

6.2.1 结果

算术运算符的结果类型由一组称做“普通算术转换”的规则确定 (C.6.3 节)。这里的整体目的就是让产生的结果具有“最大的”运算对象类型。举例来说, 如果某个二元运算符的一个运算对象是浮点数, 那么计算就将通过浮点算术完成, 结果的类型是浮点数值。如果有一个 *long* 运算对象, 那么计算就用长整数算术, 结果就是 *long*。比 *int* 小的运算对象 (例如, *bool* 和 *char*) 在运算符作用之前都将被转换到 *int*。

关系运算符, $==$ 、 $<=$ 等产生布尔值。用户定义运算符的意义及其结果类型依赖于它们的声明 (11.2 节)。

只要逻辑上可行, 一个以左值作为运算对象的运算符的结果, 仍然是指称这个左值对象的左值, 例如,

```
void f(int x, int y)
{
    int j = x = y;           // x = y 的值是赋值后 x 的值
    int* p = &++x;           // p 指向 x
    int* q = &(x++);         // 错误: x++ 不是一个左值 (它不是存储在 x 里的值)
    int* pp = &(x > y ? x : y); // 较大的那个 int 的地址
}
```

如果 $?:$ 的第二和第三个运算对象都是左值且具有相同的类型, 结果将具有这个类型而且是一个左值。以这种方式保存左值将使运算符的使用具有最大的灵活性。对于写出一些代码, 使它们能以统一而有效的方式工作在内部类型和用户定义类型上, 这种规定就特别有用处 (例如, 在写模板或者生成 C++ 代码的程序时)。

sizeof 的结果是一个无符号整型 *size_t*, 这个类型在 `<cstddef>` 里定义。指针减的结果是一个有符号的整型 *ptrdiff_t*, 也在 `<cstddef>` 里定义。

实现不必检查算术溢出, 实际上也很难做。例如,

```
void f()
{
    int i = 1;
    while (0 < i) i++;
    cout << "i has become negative!" << i << '\n';
}
```

将 (最终) 试图去增加 *i*, 使之超出最大的整数。这时会发生什么是无定义的, 但典型情况是那个值被“卷回来”变成一个负数 (在我的机器上得到 -2147483648)。与此类似, 除零的结果也是无定义的, 但这样做通常将导致程序的突然终止。特别要指出的是, 下溢、上溢和除零都不会抛出标准异常 (14.10 节)。

6.2.2 求值顺序

在一个表达式里，子表达式的求值顺序是没有定义的。特别是，你不能假定表达式从左到右求值。例如，

```
int x = f(2) + g(3); // 没定义f()或g()哪个先调用
```

不对表达式的求值顺序加以限制，使得具体实现中有可能生成更好的代码。当然，对求值顺序不加限制也可能导致无定义的结果。例如，

```
int i = 1;
v[i] = i++; // 无定义结果
```

这可能求值出 $v[1] = 1$ 或者 $v[2] = 1$ ，也可能导致更奇怪的行为。编译器可以对这样的歧义性提出警告。不幸的是，大部分编译器都不这样做。

运算符，(逗号)，&&（逻辑与）和||（逻辑或）保证了位于它们左边的运算对象一定在右边运算对象之前求值。例如， $b = (a = 2, a + 1)$ 将把3赋给 b 。关于使用&&和||的例子可以在6.2.3节找到。对于内部类型，只有在&&的第一个运算对象得到`true`的情况下才会对第二个运算对象求值；而||的第二个运算对象只是在第一个运算对象得到`false`时才会被求值；这种做法有时被称为短路求值。请注意，序列运算符，(逗号)与用于分隔函数调用参数的逗号在逻辑上是完全不同的。考虑

```
f1(v[i], i++); // 两个参数
f2((v[i], i++)); // 一个参数
```

对 $f1$ 的调用有两个参数， $v[i]$ 和 $i++$ ，而且参数求值的顺序没有定义。依赖于参数表达式的求值顺序是一种非常糟糕的风格，而且具有未予定义的行为。对 $f2$ 的调用只有一个参数，即逗号表达式 $(v[i], i++)$ ，它等价于 $i++$ 。

括号可以用做强制性的结组。例如， $a * b / c$ 表示 $(a * b) / c$ ，要得到 $a * (b / c)$ 就必须使用括号。如果用户不会发现其中的差异的话， $a * (b / c)$ 也可能被按照 $(a * b) / c$ 的方式求值。特别地，对于许多浮点计算而言， $a * (b / c)$ 和 $(a * b) / c$ 是截然不同的，所以编译器会按照写出的方式计算。

6.2.3 运算符优先级

优先级层次和结合性影响到最普通的使用情况。例如，

```
if (i <= 0 || max < i) // ...
```

的意思是“如果 i 小于等于0或者 max 小于 i ”。也就是说，它等价于

```
if ( (i <= 0) || (max < i) ) // ...
```

而不是下面这个虽然是合法的、但却是一派胡言的

```
if (i <= (0 || max) < i) // ...
```

然而，如果程序员对某些规则不很清楚，他就应该使用括号。在子表达式变得更加复杂时，使用括号会变得很普遍，但是复杂的子表达式总是错误的一个根源。因此，如果你开始感到需要括号的时候，你或许就应该考虑通过额外的变量将表达式分解开。

确实存在一些情况，其中运算符的优先级造成的结果不符合“最明显的”解释。例如，

```
if (i&mask == 0)    // 呜呼! == 表达式被作为 & 的运算对象
```

这并不是将`mask`（掩码）作用于`i`而后检测结果是否为0。因为`==`具有比`&`更高的优先级，表达式将被解释为`i & (mask == 0)`。幸运的是，编译器要想对大部分这类情况给出警告不是很难的事情。对于这个情况，加上括号是至关重要的：

```
if ((i&mask) == 0) // ...
```

值得注意的是，下面的代码不会向数学家们希望的那样工作：

```
if (0 <= x <= 99) // ...
```

这是合法的，但却将被解释为`(0 <= x) <= 99`。由于第一个比较的结果是`true`或者`false`，这个布尔值将被隐式地转换到1或0，而后用于与99比较并产生出`true`。如果真的想检测`x`是否在0 .. 99的范围内，我们就应该写：

```
if (0<=x && x<=99) // ...
```

对于新手，一个最常见的错误就是在条件中将`=`（赋值）当做`==`（相等）：

```
if (a = 7) // 呜呼! 条件中的常量赋值
```

这也很自然，因为在许多语言中确实用`=`作为“相等”。再提一句，让编译器对大部分这类错误提出警告也很容易——大部分编译器都会这样做。

6.2.4 按位逻辑运算符

按位逻辑运算符`&`、`|`、`^`、`~`、`>>`和`<<`可以应用于整型和枚举——也就是说，应用于`bool`、`char`、`short`、`int`、`long`，它们的无符号形式，以及`enum`。这时将执行常规的算术转换（C.6.3节）以确定结果类型。

按位运算符的典型用途是实现很小的集合概念（位向量）。这时，人们采用无符号整数的每个位表示集合的一个元素，位的个数是集合成员数的上限。二进制运算`&`解释为求交，`|`作为求并，`^`作为对称差，`~`作为求补。可以利用枚举作为这样一个集合的成员命名。下面是从`ostream`的某个实现中取来的一个例子：

```
enum ios_base::iostate {
    goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

流的实现可以设置和检测它的状态，比如，

```
state = goodbit;
// ...
if (state & (badbit | failbit)) // 流有问题
```

额外的括号是必须的，因为`&`具有比`|`更高的优先级。

函数在遇到输入的结束时可以采用如下方式报告情况：

```
state |= eofbit;
```

这里的`|=`运算符用于向`state`添加东西。简单赋值`state = eofbit`将清除所有其他二进制位。

从流的外面可以去查看这些流状态标志。例如，我们可以看两个流的状态有什么差异：

```
int diff = cin.rdstate() ^ cout.rdstate();    // rdstate() 返回流状态
```

计算流的差异是很常见的事情，对于其他类似的类型，计算差异也常常是最基本的工作。举例说，请想一想将一个表示正在被处理的中断集的位向量，与另一个需要去处理的中断集的位向量相比较的问题。

请注意，这里给出的有关二进制位的琐碎事物是取自*iostream*的实现，而不是想用在用户界面上。方便的位操作有可能非常重要，但是，为了可靠性、可维护性、可移植性等，还是应该将它保持在系统的底层中。有关集合的更一般概念请看标准库中的*set*（17.4.3节），*bitset*（17.5.3节）和*vector<bool>*（16.3.11节）。

位域（C.8.1节）也可以用于作为在一个机器字里移位和掩盖，以便抽取一段二进制位的一种方便方式。这些当然都可以通过按位逻辑运算符完成。例如，要抽取32位的*long*中间的16位，可以如下写：

```
unsigned short middle(long a) { return (a>>8)&0xffff; }
```

不要将按位逻辑运算符与逻辑运算符 `&&`、`||` 和 `!` 搞混了。逻辑运算符总返回*true*或*false*，它们主要是用在*if*、*while*或者*for*语句的检测部分（6.3.2节、6.3.3节）。例如，`!0`（非0）是值*true*，而 `~0`（0的补）则是一个全1的二进制模式，它在2补码表示中代表-1。

6.2.5 增量和减量

`++` 运算符用于直接表示增加，使人不必通过加和赋值的组合去间接地表示这个操作。根据定义，`++lvalue`的意思就是*lvalue* `+= 1`，这又相当于*lvalue* `= lvalue + 1`，条件是*lvalue*的求值没有副作用。指明被增加的那个对象的表达式只做一次求值。减量也类似地采用 `--` 运算符表示。运算符 `++` 和 `--` 都可以用做前缀或者后缀运算符。`++x`的值是（增量之后的）新值。例如，`y = ++x`等价于*y* `= (x += 1)`。`x++`的值则是*x*原有的值。例如，`y = x++`等价于*y* `= (t = x, x += 1, t)`，这里的*t*是一个与*x*同类型的变量[⊖]。

就像可以对指针做加和减一样，`++` 和 `--` 也可以用到对数组元素进行操作的指针上：`p++`使*p*指向下一个元素（5.3.1节）。

运算符 `++` 和 `--` 对于在循环里增加或减少变量特别有用。例如，以0结尾的字符串可以通过下面方式复制：

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++);
}
```

和C一样，C++也因为允许这类紧凑的面向表达式的编码方式而受到喜爱或者仇恨。因为

```
while (*p++ = *q++);
```

对于非C程序员而言是个小障碍，也因为以这种风格编写的代码在C和C++里绝非罕见，因此需要进一步仔细考察。

首先考虑一下更传统的复制一个字符数组的方式

```
int length = strlen(q);
for (int i = 0; i <= length; i++) p[i] = q[i];
```

⊖ 当然还要求*t*是个新变量，在环境中没有其他使用。——译者注

这确实很浪费。要确定以0结尾的字符串的长度，就需要通过读这个串去找到结尾的0。这样我们实际上就读了字符串两次：一次为确定其长度，另一次是复制。试试另一种方式：

```
int i;
for (i = 0; q[i] != 0; i++) p[i] = q[i];
p[i] = 0; // 以0结束
```

用做下标的变量*i*可以去掉，因为*p*和*q*本身就是指针：

```
while (*q != 0) {
    *p = *q;
    p++;      // 指向下一个字符
    q++;      // 指向下一个字符
}
*p = 0;      // 以0结束
```

后增量运算符使我们可以先用变量的值，而后再增加它。我们可以将循环重新写出：

```
while (*q != 0) {
    *p++ = *q++;
}
*p = 0; // 以0结束
```

由于 **p++ = *q++* 的值也就是 **q*，所以我们可以将这个例子重写为：

```
while ((*p++ = *q++) != 0) { }
```

在这个情况中，我们只有把 **q* 复制到 **p* 并增加了 *p* 之后，才会注意到它的值是不是0。因此我们就可以删除设置结束0的最后一个语句。最后，我们可以看到并不需要空的块，而且 “!= 0” 也是多余的，因为指针的结果或者整数条件实际上都要与0做比较。这样，我们就得到了开始时想找的版本：

```
while (*p++ = *q++) ;
```

这个版本比前面的版本更难读吗？对有经验的C或C++程序员而言并不是这样。在时间和空间上这个版本比前面的版本更有效吗？除了第一个调用 *strlen()* 的版本外也都不一定。哪个版本最有效，也可能依机器结构和编译器的不同而不同。

要在你的特殊计算机上复制以0结尾的字符串，最有效的方式应该用标准的串复制函数

```
char* strcpy(char*, const char*); // 来自<string.h>
```

对于更一般的复制，可以使用标准 *copy* 算法（2.7.2节、18.6.1节）。只要能用，都应该优先选用标准库的功能，而不是自己去拨弄指针和字节。标准库函数可能被在线化（7.1.1节），甚至采用特殊的机器指令实现。所以，在相信一段手工打造的代码的性能优于库函数之前，你应该先经过认真的实测。

6.2.6 自由存储

命名对象的生存时间由它的作用域（4.9.4节）决定。然而，能够建立起生存时间不依赖于建立它作用域的对象，这件事情也是很有用的。特别地，人们经常需要建立起一些对象，希望在建立它们的函数返回之后还能够使用它们。运算符 *new* 将建立起这种对象，而运算符 *delete* 能用于销毁它们。由 *new* 分配的对象被说成是在“自由存储里的”（也说是“堆对象”或者“在动态存储中分配的”）。

现在考虑我们可能如何按照写桌面计算器（6.1节）的方式写一个编译器。语法分析函数可能去构造一个为代码生成器而用的表达式树

```
struct Enode {
    Token_value oper;
    Enode* left;
    Enode* right;
    // ...
};

Enode* expr(bool get)
{
    Enode* left = term(get);
    for (;;)
        switch(curr_tok) {
            case PLUS:
            case MINUS:
            {
                Enode* n = new Enode;    // 在自由存储建立一个Enode
                n->oper = curr_tok;
                n->left = left;
                n->right = term(true);
                left = n;
                break;
            }
            default:
                return left;            // 返回结点
        }
}
```

代码生成器在此之后使用这些结点，而后销毁它们

```
void generate(Enode* n)
{
    switch(n->oper) {
        case PLUS:
            // ...
            delete n; // 删除来自自由存储的Enode
    }
}
```

由new创建的对象将一直存在，直到它被用delete显式地销毁为止。在此之后，由它所占用的存储就又能被new使用了。C++ 实现并不保证提供一个“废料收集器”，由它去找回不再存在引用的对象并使它们占用的存储重新可以被new使用。因此，我将一直假定所有通过new建立的对象都需要手工地用delete释放。如果系统里提供了废料收集器，那么在大部分情况下的delete都可以省去（C.9.1节）。

delete运算符只能用到由new返回的指针或者0，对0应用不会造成任何影响。

还可以定义运算符new的特定版本（15.6节）。

6.2.6.1 数组

也可以用new建立对象的数组。例如，

```
char* save_string(const char* p)
{
    char* s = new char[strlen(p)+1];
    strcpy(s, p);    // 从p复制到s
}
```

```

        return s;
    }

    int main(int argc, char* argv[])
    {
        if (argc < 2) exit(1);
        char* p = save_string(argv[1]);
        // ...
        delete[] p;
    }

```

“简单的” **delete** 运算符只能用于删除单个的对象，删除数组需要用 **delete[]**。

为了释放由 **new** 分配的空间，**delete** 和 **delete[]** 必须能够确定为对象分配的空间大小。这也意味着通过标准实现的 **new** 分配的对象将占用比静态对象^①稍微大一点的空间。在典型情况下需要用 一个机器字保存对象的大小。

请注意，一个 **vector**（3.7.1 节、16.3 节）是一个普通对象，因此可以用简单的 **new** 和 **delete** 分配与释放^②。例如，

```

void f(int n)
{
    vector<int>* p = new vector<int>(n);    // 单个的对象
    int* q = new int[n];                    // 数组
    // ...
    delete p;
    delete[] q;
}

```

delete[] 运算符只能应用于由 **new** 返回的到一个数组的指针或者 0，应用到 0 时不会产生任何影响。

6.2.6.2 存储耗尽

自由存储运算符 **new**、**delete**、**new[]** 和 **delete[]** 通过一些在头文件 `<new>` 里描述的函数实现（19.4.5 节）：

```

void* operator new(size_t);    // 为单个对象分配空间
void operator delete(void*);
void* operator new[](size_t); // 为数组分配空间
void operator delete[](void*);

```

当运算符 **new** 需要为某个对象分配空间时，它将调用 **operator new()** 去分配适当数量的字节。与此类似，当运算符 **new** 需要为一个数组分配空间时，就去调用 **operator new[]()**。

operator new() 和 **operator new[]()** 的标准实现并不对返回的存储做初始化。

当 **new** 无法找到需要分配的空间时会发生什么情况呢？按照默认方式，这个分配函数将抛出一个 **bad_alloc** 异常（另一种情况见 19.4.5 节）。例如，

```

void f()
{
    try {
        for(;;) new char[10000];
    }
}

```

① 通过标准实现的 **new** 创建的对象也称为动态对象，它们在动态空间里分配。与之相对的是程序里通过变量声明建立起的对象。这里所说的“静态对象”指的就是它们。——译者注

② 作者的意思是说 **vector** 不是数组，是普通对象，因此不应该用 **delete[]**。——译者注


```

    }
    catch (bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}

```

无论我们能用的空间有多少，这一程序最终都会激活**bad_alloc**处理器。

我们可以规定在存储耗尽时**new**应该去做什么。当**new**失败时，它将先去调用一个函数（如果存在），该函数是通过调用在**<new>**里声明的**set_new_handler()**设定的。例如，

```

void out_of_store()
{
    cerr << "operator new failed: out of store\n";
    throw bad_alloc();
}

int main()
{
    set_new_handler(out_of_store); // 将out_of_store()作为新的处理函数
    for (;;) new char[10000];
    cout << "done\n";
}

```

这个程序不会到达写出**done**的地方。它将写出

```
operator new failed: out of store
```

参看14.4.5节中关于**operator new()**的可能实现情况，它检查是否存在着能调用的处理函数，如果没有就抛出**bad_alloc**。**new_handler**有可能做出某些比简单地终止程序更聪明的事情。如果你知道**new**和**delete**是如何工作的——例如因为你提供了自己的**operator new()**和**operator delete()**，这个处理函数可能是企图去为**new**找到另一些存储并返回之。另一种方式是，某个用户可能提供一个废料收集器，使得**delete**的使用变成选择性的。当然，做这些事情都不是一个初学者的工作。对于所有需要自动废料收集器的人而言，应该做的正确事情是去获得一个已经写好并经过仔细测试的程序（C.9.1节）。

通过提供**new_handler**，我们就处理了在程序中所有**new**的常规使用中对存储耗尽情况的检查也存在另外两种控制存储分配的替代方法。我们可以为**new**的非标准使用提供非标准的分配函数和释放函数（15.6节），或者是依靠用户提供有关存储分配的附加信息（10.4.11节、19.4.5节）。

6.2.7 显式类型转换

有时我们需要处理“原始的存储”，也就是那种保存或者将要保存某种对象的存储，而编译器并不知道对象的类型。例如，一个存储分配程序可能返回一个新分配存储块的**void***指针，或者我们希望说明应该把某个整数当做一个I/O设备的地址等：

```

void* malloc(size_t);

void f()
{
    int* p = static_cast<int*>(malloc(100)); // 新分配的存储，用做一些整数
    IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00); // 位于0Xff00的设备
    // ...
}

```

编译器并不知道由`void*`所指的对象的类型，它也不可能知道`0Xff00`是否为合法的地址。因此，这些转换的正确性完全掌握在程序员的手里。显式的类型转换常常被称做强制，这只在很少的情况下是绝对必要的东西。然而，在传统上它却被过度使用，并成为一个主要的错误根源。

`static_cast`运算符完成相关类型之间的转换，例如在同一个类层次结构中的一个指针类型到另一个指针类型，整型到枚举类型，或者浮点类型到整型等。`reinterpret_cast`处理互不相关类型之间的转换，例如从整型到指针，或者从一个指针到另一个毫不相干的指针类型。这种区分使编译器能对`static_cast`做某种小的类型检查，并使程序员能看清由`reinterpret_cast`的存在所表明更危险的类型转换。有些`static_cast`是可移植的，但极少`reinterpret_cast`能是这样。对`reinterpret_cast`很难做出任何保证，但一般而言，它提供了一个新类型的值，并保持其参数原来的二进制模式。如果目标类型具有至少与原值同样多的二进制位，我们就可以将结果再通过`reinterpret_cast`，转回到原来的类型并使用它。只是在被使用的类型正好就是所定义值原来的类型时，才能保证`reinterpret_cast`的结果是可以用的。

如果你感到要去使用显式的类型转换，请花一点时间去考虑它是否确实有必要。对于C必须用（1.6节），或者在C++的早期版本里需要用（1.6.2节、B.2.3节）的显式类型转换中的大部分情况，在C++里都不再需要了。在许多程序里完全可以避免显式的类型转换；在另一些程序里，其使用也可以局部到少数例行程序内部。在这本书里，在实际情况中出现显式类型转换的只有6.2.7节、7.7节、13.5节、13.6节、17.6.2.3节、15.4节、25.4.1节和E.3.1节。

·另外还提供了一种运行中检查的转换形式`dynamic_cast`（15.4.1节），还有一种清除`const`和`volatile`限定符的转换形式`const_cast`（15.4.2.1节）。

C++从C那里继承来 $(T)e$ 记法，这种形式可以执行能用`static_cast`、`reinterpret_cast`和`const_cast`的组合表述的任何转换，它从表达式`e`出发去做出一个`T`类型的值（B.2.3节）。这种C风格的强制远比所有上述的命名转换更加危险，因为在大的程序里这种记法极难看清楚，也因为程序员并没有将转换的意图明确表示出来。也就是说， $(T)e$ 可能做相关类型间的可移植转换，无关类型间的不可移植转换，或者去掉指针类型的`const`修饰符。如果不知道`T`和`e`的准确类型，你什么都说不清楚。

6.2.8 构造函数

从值`e`构造出一个类型`T`的值可以用函数记法 $T(e)$ 表述。例如，

```
void f(double d)
{
    int i = int(d);           // 截断d
    complex z = complex(d);  // 从d做出一个complex
    // ...
}
```

这种 $T(e)$ 结构有时被称做函数风格的强制。不幸的是，对于内部类型`T`而言， $T(e)$ 等价于 $(T)e$ ，这也意味着 $T(e)$ 对于许多内部类型是不安全的。对于算术类型，值的转换可以出现截断，甚至从较长的整数类型到较短（如从`long`到`char`）的显式转换也可能导致不可移植的依赖于具体实现的行为。我将只在一个值的构造有良好定义的地方使用这种记法，即用于算术类型之间的缩窄转换（C.6节），从整数到枚举的转换（4.8节）和用户定义类型的对象构造（2.5.2节、10.2.3节）。

指针转换不能直接采用 $T(e)$ 的记法形式表示。例如，`char*(2)`是一个语法错误。让构造

函数的记法回避这样的一类危险转换是件很好的事情。但不幸的是，这种保护却会被对指针使用`typedef`产生的名字（4.9.7节）所打破。

构造函数记法`T()`用于描述类型`T`的默认值。例如，

```
void f(double d)
{
    int j = int();           // 默认的int值
    complex z = complex();   // 默认的complex值
    // ...
}
```

对于内部类型而言，这样显式地使用构造函数，得到的是将`0`转换到该类型所得到的值（4.9.5节）。这样，`int()`也就是写`0`的另一种方式。对于用户定义类型`T`，如果存在这种函数的话，`T()`由`T`的默认构造函数定义（10.4.2节）。

对于内部类型也能使用构造函数记法，这一点在写模板的时候就特别重要。在这种情况下，程序员无法知道一个模板参数是内部类型还是用户定义类型（16.3.4节、17.4.1.2节）。

6.3 语句概览

这里是C++ 语句的一览表和某些例子：

语句的语法
<pre>statement: declaration { statement-list_{opt} } try { statement-list_{opt} } handler-list expression_{opt} ; if (condition) statement if (condition) statement else statement switch (condition) statement while (condition) statement do statement while (expression) ; for (for-init-statement condition_{opt} ; expression_{opt}) statement case constant-expression : statement default : statement break ; continue ; return expression_{opt} ; goto identifier ; identifier : statement statement-list: statement statement-list_{opt} condition: expression type-specifier declarator = expression handler-list: catch (exception-declaration) { statement-list_{opt} } handler-list handler-list_{opt}</pre>

请注意，声明也是一种语句。这里没有列出赋值语句或过程调用语句，因为赋值和函数调用都是表达式。用于处理异常的语句，*try*块，将在8.3.1节描述。

6.3.1 声明作为语句

一个声明也是一个语句。除非一个变量被声明为*static*，否则它的初始式的执行就将在控制线程经过这个声明的时候进行（10.4.8节）。允许在所有写语句的地方（以及几处其他地方；6.3.2.1节、6.3.3.1节）写声明，是为了使程序员能够最大限度地减少由于未初始化的变量而导致的错误，并使代码得到更好的局部化。很少有某种理由说，必须在还没有某变量应该保存的值之前引进这个变量。例如，

```
void f(vector<string>& v, int i, const char* p)
{
    if (p==0) return;
    if (i<0 || v.size()<=i) error("bad index");
    string s = v[i];
    if (s == p) {
        // ...
    }
    // ...
}
```

能将变量放在可执行的代码之后，对于写许多常量以及程序设计的单赋值风格（其中对象的值在初始化之后就不再改变）都是最基本的。对于用户定义类型而言，将变量的定义推迟到适当的初始式已经可以使用之时，还可以导致更好的执行性能。例如，

```
string s; /* ... */ s = "The best is the enemy of the good.";
```

很容易比下面的语句慢得多

```
string s = "Voltaire";
```

也有一些情况中需要声明一个没有初始式的变量，最常见的就是需要用语句去初始化它。这方面的例子如输入用的变量和数组等。

6.3.2 选择语句

一个值可以被*if*语句或者*switch*语句检测：

```
if ( condition ) statement
if ( condition ) statement else statement
switch ( condition ) statement
```

比较运算符

```
==  !=  <  <=  >  >=
```

在比较结果为真时返回*bool*值*true*；否则就返回*false*。

对于*if*语句，如果条件表达式（*condition*）非0则（只）执行第一个语句（*statement*），否则就（只）执行第二个语句。这也意味着任何算术或者指针表达式都可以用做条件。例如，如果*x*是个整数，那么

```
if (x) // ...
```

的意思就是

```
if (x != 0) // ...
```

对于指针 p

```
if (p) // ...
```

就是直接检测“ p 是指向一个合法对象”（假定 p 经过正确的初始化）的语句，而

```
if (p != 0) // ...
```

则是通过与一个明确地不指向任何对象的值相比较，从而间接地陈述了同一个问题。请注意，指针0的表示未必在所有机器上都是全0（5.1.1节）。但我检查过的所有编译器对于这两种写法生成的代码都完全一样。

逻辑运算符

```
&&  ||  !
```

最经常被用在条件里。运算符 $\&\&$ 和 $||$ 除了在必要之时，是不会去对其第二个运算对象求值的。例如，

```
if (p && l < p->count) // ...
```

首先检查 p 是否非0，只有在 p 非0的情况下才检测 $l < p->count$ 。

某些if语句可以很方便地用条件表达式取代。例如，

```
if (a <= b)
    max = b;
else
    max = a;
```

写成下面形式更好些：

```
max = (a <= b) ? b : a;
```

括着条件表达式的括号并不必要，但我觉得，写了它们可以使代码更容易读一些。

switch语句可以作为写一组if语句的另一种方式。例如，

```
switch (val) {
case 1:
    f();
    break;
case 2:
    g();
    break;
default:
    h();
    break;
}
```

也可以等价地表述为

```
if (val == 1)
    f();
else if (val == 2)
    g();
```

```
else
    h();
```

两者意义相同，但是第一个（用`switch`的）版本更好一些，因为操作的性质（相对于一组常量检测一个值）表现得更明显。这也使对于一些比较复杂的例子而言，`switch`语句更容易阅读。它也可能导致生成更好的代码。

请当心，`switch`语句中的每个case必须以某种方式终止，除非你希望让执行直接进入下一个case中。考虑

```
switch (val) {           // 当心
case 1:
    cout << "case 1\n";
case 2:
    cout << "case 2\n";
default:
    cout << "default: case not found\n";
}
```

如果`val == 1`，它将打印出

```
case 1
case 2
default: case not found
```

这会把没准备的人吓了一跳。给那些（罕见的）让执行掉进下一个case的情况加上注释，说明这样做是有意的，这是一个很好的建议，也使无意中掉下去被假定为一个错误。最常见的结束case的方式是用`break`语句，但是`return`也常常被使用（6.1.1节）。

6.3.2.1 在条件中的声明

为了避免意外地错误使用变量，在最小的作用域里引进变量是一个很好的想法。特别地，最好是把局部变量的定义一直推迟到可以给它初始值时再去做。采用这种方式，就不会出现因为使用未初始化的变量而造成的麻烦了。

这两条原则的一个最优雅的应用就是在条件中声明变量。考虑

```
if (double d = prim(true)) {
    left /= d;
    break;
}
```

在这里，`d`被声明和初始化，初始化后的`d`值又被作为条件的值。`d`的作用域从它的声明点一直延伸到这个条件所控制的语句的结束。例如，如果这个`if`语句有`else`分支，在这两个分支里，`d`都处于作用域之中。

另一种最明显的传统方式是在条件之前声明`d`。但是这就打开了使用`d`的作用域：在其初始化之前，以及在预想中它有价值的生存阶段之后：

```
double d;
// ...

d2 = d;    // 呜呼!
// ...

if (d = prim(true)) {
    left /= d;
```

```

    break;
}
// ...

d = 2.0; // d的两个不相关的使用

```

在条件中声明变量，除了逻辑方面的优点之外，还能产生出更紧凑的源代码。

在条件中的声明只能声明和初始化单个的变量或者**const**。

6.3.3 迭代语句

循环可以用**for**、**while**或者**do**语句表述

```

while ( condition ) statement
do statement while ( expression );
for ( for-init-statement conditionopt; expressionopt ) statement

```

这些语句中的每一个都将反复执行一个称为受控语句或者循环体的语句（**statement**），直到条件（**condition**）变成假，或者程序员要求以其他方式跳出这个循环。

for语句是为了表述最规范的循环形式。循环变量、结束条件（**condition**），以及更新循环变量的表达式（**expression**）都可以在“最前面”的一行里描述。这样可以极大地提高可读性，也会减少出错的频度。如果不需要初始化，初始化语句可以为空。如果忽略了条件部分，那么这个**for**语句就将永远循环下去，除非用户明确地通过**break**、**return**、**goto**、**throw**或者某些不那么明显的方式，例如调用**exit()**（9.4.1.1节），从循环中退出来。如果忽略了表达式部分，我们就必须在循环体里以某种方式更新循环变量。如果一个循环不是简单的“引进一个循环变量、检测条件、更新循环变量”类型的，那么最好是用**while**语句描述。**for**语句也经常被用于描述没有明确结束条件的循环：

```

for(;;) { // “永远”
    // ...
}

```

一个**while**语句简单地执行受其控制的语句，直到它的条件变成**false**。对于那些没有明确的循环变量，或者对循环变量的更新自然地出现在循环体的中间时，我更偏向于采用**while**语句而不是**for**语句。输入循环就是一种没有明确的循环变量的循环：

```

while(cin>>ch) // ...

```

按照我的经验，**do**语句是错误和混乱的一个根源。究其原因，缘于它在条件求值之前总要执行循环体一次。然而，要使循环体正确工作，一定会有一些很像条件的东西必须在第一次到达时就成立。但与我的揣测不同，我发现更经常的情况是这种条件并不像所期望的那样成立，无论是在程序第一次写出和调试时，还是在它前面的代码被修改之后。我也更喜欢有关条件“位于前面，使我能够看到它”。由于这些，我尽量避免**do**语句。

6.3.3.1 **for**语句里的声明

可以在**for**语句的初始化部分中声明变量。如果这个初始化部分是一个声明，它所引进的变量（一个或一些）直到**for**语句的结束都处于作用域之中。例如，

```

void f(int v[], int max)
{
    for (int i = 0; i < max; i++) v[i] = i * i;
}

```

如果在退出`for`循环之后还需要知道下标的最终值，那就必须在`for`循环之外声明这个下标变量（例如6.3.4节）。

6.3.4 `goto`

C++ 拥有臭名昭著的`goto`语句：

```
goto identifier ;
identifier : statement
```

`goto`在高级程序设计中极少有用，但是在那些不是由人写出，而是由某个程序生成出来的C++代码里就可能很有用处。例如，`goto`可能被用在一个由分析程序生成器生成语法的分析程序里。在一些罕见的优化性能极端重要的程序里，`goto`也可能非常重要，例如在某些实时应用的内层循环中。

标号（`label`）的作用域是它所在的那个函数。这就意味着你能够利用`goto`跳进或者跳出一个块。仅有的限制就是你不能跳过初始式，也不能跳进异常处理程序（8.3.1节）。

在常规代码中极少存在`goto`的有意义应用，这种情况之一是从嵌套的循环或者`switch`语句中退出来（`break`语句只能退出最内层的循环或者`switch`语句）。例如，

```
void f()
{
    int i;
    int j;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++) if (nm[i][j] == a) goto found;
    // not found
    // ...
found:
    // nm[i][j] == a
}
```

还有一种`continue`语句，实际上，它就是跳到一个循环语句的最后，如6.1.5节的解释。

6.4 注释和缩进编排

明智地使用注释，一致性地使用缩进编排形式，可以使阅读和理解一个程序的工作变得更愉快些。人们经常采用的有几种不同的一致性缩进编排风格。我看不出偏爱某种风格胜过另一种的明显理由（虽然像大多数程序员一样，我也有我的偏爱，本书就反映了有关的情况）。这一点也同样适用于注释风格。

注释可能被人们以许多方式误用，并因此严重地影响程序的可读性。编译器当然不会理解注释的内容，因此它无法保证一个注释

[1] 是有意义的。

[2] 描述了这个程序。

[3] 是符合目前情况的。

许多程序里都包含着不易理解的、歧义的，或者根本就是错误的注释。糟糕的注释还不如没有注释。

如果某些东西已经由语言本身说明白，那么就不应当再作为注释中提及的内容。这个说

法针对的就是下面这一类东西：

```
// 变量“v”必须初始化
// 变量“v”只能由函数“f()”使用
// , 在调用这个文件中任何其他函数之前调用函数“init()”
// 在你的程序最后调用函数“cleanup()”
// 不要使用函数“weird()”
// 函数“f()”有两个参数
```

如果是正常地使用C++，这种注释一般都应该认为是不必要的。例如，人们可以根据连接规则（9.2节）以及类的可见性、初始化和清理规则（10.4.1节）来说明上面的注释例子完全是多余的。

一旦某些东西已经在语言里说清楚了，就不应该在注释里第二次提它。例如，

```
a = b + c; // a变为b + c
count++; // count加1
```

这种注释比多余还要坏。它们增加了读者必须去看的正文数量，也使程序结构变得更模糊。而且它们还可能是错的。注意，无论如何，这类注释被广泛地用在程序设计语言的教科书里，例如这一本书里。这也是在教科书里的程序与真实程序之间的众多差异之一。

我的偏爱在于：

- [1] 为每一个源文件写一个注释，一般性地陈述在它里面有哪些声明，对有关手册的引用，为维护而提供的一般性提示，如此等等。
- [2] 对每个类、模板和名字空间写一个注释。
- [3] 对每个非平凡的函数写一个注释，陈述其用途，所用的算法（除非算法非常明显），以及可能有的关于它对于环境所做的假设。
- [4] 对每个全局的和名字空间的变量和常量写一个注释。
- [5] 在非明显和/或不可移植的代码处的少量注释。
- [6] 极少其他的东西。

例如，

```
//  tbl.c: Implementation of the symbol table.
/*
    Gaussian elimination with partial pivoting.
    See Ralston: "A first course ..." pg 411.
*/

//  swap() assumes the stack layout of an SGI R6000.
/* *****

    Copyright (c) 1997 AT&T, Inc.
    All rights reserved

    ***** /
```

一组经过良好选择和良好书写的注释是好程序中最具本质性的一个组成部分。写出好注释可能就像写出好程序一样困难，但这是一种值得去好好修养的艺术。

请再次注意, 如果在一个函数里仅仅使用 `//` 形式的注释, 那么这函数的任何部分都可以通过 `/* */` 风格的注释去掉, 反过来也一样。

6.5 忠告

- [1] 应尽可能使用标准库, 而不是其他的库和“手工打造的代码”; 6.1.8节。
- [2] 避免过于复杂的表达式; 6.2.3节。
- [3] 如果对运算符的优先级有疑问, 加括号; 6.2.3节。
- [4] 避免显式类型转换(强制); 6.2.7节。
- [5] 若必须做显式类型转换, 提倡使用特殊强制运算符, 而不是C风格的强制; 6.2.7节。
- [6] 只对定义良好的构造使用 `T(e)` 记法; 6.2.8节。
- [7] 避免带有无定义求值顺序的表达式; 6.2.2节。
- [8] 避免 `goto`; 6.3.4节。
- [9] 避免 `do` 语句; 6.3.3节。
- [10] 在你已经有了去初始化某个变量的值之前, 不要去声明它; 6.3.1节、6.3.2.1节、6.3.3.1节。
- [11] 使注释简洁、清晰、有意义; 6.4节。
- [12] 保持一致的缩进编排风格; 6.4节。
- [13] 倾向于去定义一个成员函数 `operator new()` (15.6节) 去取代全局的 `operator new()`; 6.2.6.2节。
- [14] 在读输入的时候, 总应考虑病态形式的输入; 6.1.3节。

6.6 练习

1. (*1) 将下面的 `for` 循环重写为采用 `while` 循环的等价形式:

```
for (i=0; i<max_length; i++) if (input_line[i] == '?') quest_count++;
```

重写这个片段, 用一个指针作为被控制变量, 其检测采用 `*p == '?'` 的形式。

2. (*1) 为下面各个表达式加上全部括号:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1, 2) + 3
a = -1 ++ b -- - 5
a = b == c ++
a = b = c = 0
a[4][2] *= *b ? c : *d * 2
a-b, c=d
```

3. (*2) 读入一系列由空白分隔的(名字, 值)对, 其中每个名字是由空白分隔的一个单词, 值是一个整数或者一个浮点值。计算并打印出对应于每个名字的所有值之和与平均值, 以及所有名字的和与平均值。提示: 6.1.8节。
4. (*1) 写出一个表格, 其中列出以各种可能的 `0` 和 `1` 组合作为运算对象, 进行按位逻辑运算所得到 (6.2.4节) 的值。

5. (*1.5) 找出5种不同的其意义无定义的C++ 结构 (C.2节)。(*1.5) 找出5种不同的其意义由实现确定的C++ 结构 (C.2节)。
6. (*1) 找出10个不可移植的C++ 代码的例子。
7. (*2) 写出5个表达式, 对它们的求值顺序没有定义。执行它们, 看看某一个或 (最好) 多个实现对它们怎么做。
8. (*1.5) 如果你在你的系统上除0会发生什么事情? 在上溢和下溢时又会怎么样?
9. (*1) 给下面各表达式加上全部括号:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) 写出下面函数: *strlen()*, 它返回C风格字符串的长度; *strcpy()*, 它将一个C风格字符串复制到另一个; *strcmp()*, 它比较两个C风格的字符串。考虑参数类型和返回值类型应该是什么。而后将你的函数与在 *<cstring>* (*<string.h>*) 里以及在20.1.4节描述的标准库版本做一个比较。
11. (*1) 看看你的编译器对下面这些错误有何反应:

```
void f(int a, int b)
{
    if (a = 3) // ...
    if (a&077 == 0) // ...
        a := b+1;
}
```

设计一些更简单的错误, 并看看编译器如何反应。

12. (*2) 修改6.6[3] 的程序, 使之同时计算出中间值。
13. (*2) 写一个函数*cat()*, 它取两个C风格字符串为参数, 返回一个字符串, 该字符串是两个参数串的拼接。利用*new*为这个结果取得存储。
14. (*2) 写一个函数*rev()*, 它取一个C风格字符串*p*为参数, 并反转其中的字符。也就是说, 在*rev(p)* 之后*p*的最后一个字符将变成第一个, 如此等等。
15. (*1.5) 下面例子将做些什么?

```
void send(int* to, int* from, int count)
// Duff设施, 有帮助的注释被有意删去了
{
    int n = (count+7)/8;
    switch (count%8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
    } while (--n>0);
    }
}
```

为什么会有人想写这样的东西？

16. (*2) 写一个函数 `atoi(const char*)`，它以一个包含数字的C风格字符串为参数，返回与之对应的 `int` 值。例如，`atoi("123")` 应是 `123`。修改 `atoi()`，使之除了能处理简单的十进制数之外，还能处理C++ 的八进制和十六进制记法形式。修改 `atoi()` 以处理C++ 的字符常量记法。
17. (*2) 写一个函数 `itoa(int i, char b[])`，它在 `b` 中建立起 `i` 的字符串表示并返回 `b`。
18. (*2) 键入计算器的例子并使之能够工作。不要“节约时间”去使用已有的正文文件。你将会从发现并改正各种“小而蠢的错误”中学到许多东西。
19. (*2) 修改计算器程序，使它能够在报告错误时给出相应的行号。
20. (*3) 设法使用户可以在计算器中定义函数。提示：将一个函数定义为运算的一个序列，就像用户键入它们那样。这种序列可以存储为字符串，或者存储为单词的表。当函数被调用时，就读取并执行这些运算。如果你希望用户定义函数能够使用参数，那么你就必须为此去发明一种记法形式。
21. (*1.5) 改造桌面计算器，让它使用一种 `symbol` 结构，而不是使用静态变量 `number_value` 和 `string_value`。
22. (*2.5) 写一个程序，使它能剥掉C++ 程序里的所有注释。即，从 `cin` 读入，删除所有的 `//` 和 `/* */` 注释，而后将其写到 `cout`。不要费力去把输出的布局弄得比较好看（那可以成为另外一个更困难的练习题），也不要顾虑错误的程序。请留意位于注释、字符串和字符常量中的 `//`、`/*` 和 `*/`。
23. (*2) 去找一些程序，取得一些有关在实际中使用的缩进编排、命名和注释的认识。

第7章 函 数

迭代的是人，
递归的是神。

——L. Peter Dautsch

函数声明和定义——参数传递——返回值——函数重载——歧义性解析——默认参数
——*stdargs*——指向函数的指针——宏——忠告——练习

7.1 函数声明

在C++ 程序里，完成某件工作的一种典型方式就是调用一个函数去做那件事情。定义函数是你刻画怎样完成某个操作的一种方式。一个函数只有在预先声明之后才能调用。

在一个函数声明中，需要给出函数的名字，这个函数返回的值的类型（如果有的话），以及在调用这个函数时必须提供的参数的个数和类型。例如，

```
Elem* next_elem();  
char* strcpy(char* to, const char* from);  
void exit(int);
```

参数传递的语义等同于初始化的语义。参数的类型被逐个检查，如果需要就会做隐式的参数类型转换。例如，

```
double sqrt(double);  
double sr2 = sqrt(2);           // 以参数double(2) 调用sqrt()  
double sq3 = sqrt("three");    // 错误：sqrt() 要求类型为double的参数
```

这种检查和类型转换的价值绝不应该低估。

在函数声明中可以包含参数的名字。这样做可能对读程序的人有所帮助，但编译器将简单地忽略掉这样的名字。正如在4.7节所提到的，以**void**作为返回值类型表示这个函数不返回值。

7.1.1 函数定义

在程序里调用的每个函数都必须在某个地方定义（仅仅一次）。一个函数定义也就是一个给出了函数体的函数声明。例如，

```
extern void swap(int*, int*); // 声明  
void swap(int* p, int* q)    // 定义  
{  
    int t = *p;  
    *p = *q;  
    *q = t;  
}
```

一个函数的定义和对它的所有声明必须都描述了同样的类型。不过，这里并不把参数名字作为类型的一部分，因此参数名不必保持一致。

在函数的定义里，可以存在不使用的参数，这也是常能看到的情况：

```
void search(table* t, const char* key, const char*)
{
    // 第三个参数没有使用
}
```

如上所示，根本不使用的参数可以采用不予命名的方式明示。典型情况是，出现未命名参数的原因是做过代码的简化，或者是计划在将来做功能扩充。对于这两种情况，虽然不使用但还是让参数留在那里，就能保证那些调用函数的地方不会受到修改的影响。

函数可以定义为**inline**（在线的）。例如，

```
inline int fac(int n)
{
    return (n<2) ? 1 : n*fac(n-1);
}
```

inline描述符是给编译器的一个提示，要求它试着把所有对**fac()**的调用在线化，而不是仅仅一次放好函数的代码，而后通过正常的函数调用机制调用这段代码。一个聪明的编译器也可能直接对调用**fac(6)**产生出常量**720**。互相递归的在线函数、自递归的^①或者并不依赖于输入的在线函数等的存在，使得保证所有**inline**函数的调用都能在线化是根本不可能的。对编译器的聪明程度并没有做任何明确规定，所以对于上例，某个编译器可能产生**720**，另一个可能是**6*fac(5)**，再有一个产生出完全没有在线化的**fac(6)**。

要在不存在特别聪明的编译器和连接机制的情况下使在线化有可能进行，在线函数的定义——而不仅仅是它的声明——就必须在作用域里（9.2节）。**inline**描述符并不影响函数的语义。特别地，每个在线函数仍然会有自己的独立地址，在线函数里的那些**static**变量（7.1.2节）也将有自己的地址。

7.1.2 静态变量

局部变量将在运行线程达到其定义时进行初始化。按照默认方式，这件事发生在函数的每次调用中，且函数的每个调用有自己的一份局部变量副本。如果一局部变量被声明为**static**，那么将只有惟一的一个静态分配的对象（C.9节），它被用于在该函数的所有调用中表示这个变量。这个对象将只在执行线程第一次到达它的定义时初始化。例如，

```
void f(int a)
{
    while (a--) {
        static int n = 0;           // 初始化一次
        int x = 0;                  // 在每个f()调用时初始化a次
        cout << "n == " << n++ << ", x == " << x++ << '\n';
    }
}

int main{}
```

① 估计作者指的是例如**int f(int n) { return f(n); }**一类的函数。——译者注

```
{
    f(3);
}
```

这将打印出

```
n == 0, x == 0
n == 1, x == 0
n == 2, x == 0
```

静态变量为函数提供了一种“存储器”，使我们不必去引进可能被其他函数访问或破坏的全局变量[⊖]（另见10.2.4节）。

7.2 参数传递

当一个函数被调用时，将安排好其形式参数所需要的存储，各个形式参数将用对应的实际参数进行初始化。参数传递的语义与初始化的语义完全相同。特别是，需要对照着每一个形式参数检查与之对应的实际参数的类型，并执行所有标准的或者用户定义的类型转换。另有一些特殊规则处理数组参数的传递（7.2.1节），一种传递不加检查的参数的机制（7.6节）以及一种刻画默认参数的机制（7.5节）。考虑

```
void f(int val, int& ref)
{
    val++;
    ref++;
}
```

当`f()`被调用时，`val++`增加的是第一个实际参数的一个局部副本，而`ref++`增加的是第二个实际参数本身。例如，

```
void g()
{
    int i = 1;
    int j = 1;
    f(i, j);
}
```

将使`j`增加1，而`i`并不增加。第一个参数`i`传递的是值，而第二个参数`j`传递的是引用。正如第5.5节里所说，修改引用参数的函数将会使程序更难阅读，因此最好避免写这种函数（但也请看看第21.3.2节）。当然也应注意到，通过引用方式传递大的对象，比通过值传递的效率更高一些。在这种情况下，可以将有关的参数声明为`const`，以指明使用这种参数仅仅是为了效率的原因，而不是想让调用函数能够修改对象的值：

```
void f(const Large& arg)
{
    // 如果没有显式地做类型转换，“arg”的值就不能修改
}
```

如果在一个引用参数的声明中没有`const`，就应该认为，这是想说明该参数将被修改：

⊖ 这里所说的“存储”是指不依赖于函数调用而独立存在的存储机制。静态变量在一个函数里定义，只能在这个函数里通过变量名访问，但它的存在并不依赖于函数的调用。因此，这种变量可以用于保存需要一直存在，以便在这个函数内部使用的信息。——译者注

```
void g(Large& arg); // 假定g()修改arg
```

与此类似，将指针参数声明为`const`，也就是告知读者，函数将不修改由这个参数所指的对象。例如，

```
int strlen(const char*);           // 求C风格的字符串的长度
char* strcpy(char* to, const char* from); // 复制C风格的字符串
int strcmp(const char*, const char*); // 比较C风格的字符串
```

使用`const`参数的重要性将随着程序的规模增大而进一步增长。

请注意，参数传递的语义不同于赋值的语义。对于`const`参数、引用参数和某些用户定义类型的参数（10.4.4.1节），这一点都非常重要。

文字量、常量和需要转换的参数都可以传递给`const&`参数，但不能传递给非`const`的引用参数。允许对`const T&`参数进行转换，就保证了对这种参数所能提供的值集合，正好与通过一个临时量传递`T`参数的集合相同。例如，

```
float fsqrt(const float&); // Fortran风格的sqrt采用引用参数

void g(double d)
{
    float r = fsqrt(2.0f); // 传递的是保存2.0f的临时量的引用
    r = fsqrt(r);          // 传递r的引用
    r = fsqrt(d);          // 传递的是保存float(d)的临时量的引用
}
```

对非`const`引用参数不允许做类型转换（5.5节），这种规定能帮助我们避免了一种由于引入临时量而产生的可笑错误。例如，

```
void update(float& i);

void g(double d, float r)
{
    update(2.0f); // 错误：const参数
    update(r);    // 传递r的引用
    update(d);    // 错误：要求类型转换
}
```

如果允许所有这些调用，`update()`将会不声不响地去更新一个马上就会被删除的临时量^①。这经常会让程序员极不愉快地大吃一惊。

7.2.1 数组参数

如果将数组作为函数的参数，传递的就是到数组的首元素的指针。例如，

```
int strlen(const char*);

void f()
{
    char v[] = "an array";
    int i = strlen(v);
    int j = strlen("Nicholas");
}
```

① 作者指的是最后那个函数调用。请读者自己分析一下，在作者的假设下将会发生什么情况，就不难看清楚这一点了。——译者注

也就是说, 类型 $T[]$ 作为参数传递时将被转换为一个 T^* 。这也就意味着, 对数组参数的某个元素的赋值, 将改变实际参数数组中那个元素的值。换句话说, 数组与其他类型不同, 数组不会 (也不能) 按值的方式传递。

对于被调用函数而言, 数组参数的大小是不可用的。这可能成为一个麻烦, 但存在许多可以绕过这个问题的方式。C风格的字符串是以0结束的, 它们的大小就很容易计算。对于别的数组, 可以传递第二个参数, 用它刻画数组的大小, 例如,

```
void compute1(int* vec_ptr, int vec_size);    // 一种方法

struct Vec {
    int* ptr;
    int size;
};

void compute2(const Vec& v);                // 另一种方法
```

换一种方式, 我们也可以不用数组, 而用 `vector` (3.7.1节、16.3节) 等类型。

多维数组的情况更加诡异 (C.7节), 但通常可以改用指针的数组, 这样做并不需要特别的处理。例如,

```
char* day[] = {
    "mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

再提一次, `vector` 等类型可以用于代替内部的、低级的数组和指针等一类东西。

7.3 返回值

一个没有声明为 `void` 的函数都必须返回一个值 (然而 `main()` 是特殊的; 3.2节)。与此相反, `void` 函数就不能返回值。例如,

```
int f1() { }                // 错误: 缺返回值
void f2() { }               // ok

int f3() { return 1; }      // ok
void f4() { return 1; }     // 错误: 在void函数里返回值

int f5() { return; }        // 错误: 缺返回值
void f6() { return; }       // ok
```

返回值由返回语句描述。例如,

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

一个调用自己的函数被称为是递归的。

函数里可以出现多个返回语句:

```
int fac2(int n)
{
    if (n > 1) return n*fac2(n-1);
    return 1;
}
```

就像参数传递的语义一样, 函数返回值的语义也与初始化的语义相同, 可以认为返回语句所做的就是去初始化一个具有返回类型的匿名变量。这时将对照函数的返回类型检查返回表达式的类型, 并执行所有标准的或用户定义的转换。例如,

```
double f() { return I; } // I被隐式地转换到double(I)
```

每当一个函数被调用时，就会建立起它的所有参数和局部（自动）变量的一套新副本。在该函数返回后，这些存储又会被另做他用。所以，绝不能返回指向局部变量的指针，因为被指位置中内容的改变情况是无法预料的：

```
int* fp() { int local = I; /* ... */ return &local; } // 错
```

这种错误一般没有通过引用造成的类似错误那么普遍：

```
int& fr() { int local = I; /* ... */ return local; } // 错
```

幸运的是，编译器很容易对返回局部变量的引用提出警告。

void函数不能返回值。当然，由于对**void**函数的调用不会产生任何值，所以，一个**void**函数可以将另一个**void**函数作为它的**return**语句中的表达式。例如，

```
void g(int* p);
```

```
void h(int* p) { /* ... */ return g(p); } // 可以：返回“无值”
```

这种返回形式有其重要性，如果要写的模板函数的返回类型是模板参数，就很可能需要用这种东西（18.4.4.2节）。

7.4 重载函数名

最常见的情况是，应该给不同函数以不同的名字。但是，当某些函数在不同类型的对象上执行概念上相同的工作时，能给它们取相同的名字就更方便了。将同一个名字用于在不同类型上操作的函数的情况称为重载。这一技术早已用于C++中的基本运算。例如，对于加法只存在一个名字，+，然而它却可以用于做整数、浮点数和指针值的加法。这种思想很容易扩充到程序员定义的函数。例如，

```
void print(int);           // 打印int
void print(const char*); // 打印C风格的字符串
```

如果只考虑编译器，那么具有同样名字的函数之间共同的东西就只是这个名字。按照推测，这些函数应该在某种意义上是相互类似的，但语言无法在这方面限制或者帮助程序员。这样，重载函数名从根本上说就是一种记法上的方便。对那些有着习用名字的函数，如**sqrt**、**print**和**open**等，这种方便性是很重要的。当一个名字在语义上很重要时，这种方便将是根本性的。这种情况出现在例如+、*和<<等运算符，构造函数（11.7节），以及通用型程序设计（2.7.2节、第18章）等许多情况中。当一个函数**f**被调用时，编译器就必须弄清究竟应该调用具有名字**f**的哪一个函数。为了完成这项工作，它需要将实际参数的类型与所有名字为**f**的函数的形式参数的类型相比较。基本想法是去调用其中的那个在参数上匹配得最好的函数，如果不存在匹配得最好的函数，就给出一个编译错误。例如，

```
void print(double);
void print(long);

void f()
{
    print(1L);      // print (long)
    print(1.0);     // print (double)
    print(1);       // 错误，歧义性：print(long(1)) 或 print(double(1))？
}
```

要从一集重载的函数中找到应实际调用的那个正确版本，就需要找到在参数表达式的类型和函数的（形式）参数类型之间的最好匹配。为了尽可能接近我们关于怎样做最合理的观念，需要按顺序检查下面的一系列判断准则：

- [1] 准确匹配；也就是说，无须任何转换或者只须做平凡转换（例如，数组名到指针，函数名到函数指针，*T*到*const T*等）的匹配。
- [2] 利用提升的匹配；即包括整数提升（*bool*到*int*，*char*到*int*，*short*到*int*以及它们的无符号版本；C.6.1节）以及从*float*到*double*的提升。
- [3] 利用标准转换（例如，*int*到*double*，*double*到*int*，*double*到*long double*，*Derived**到*Base**（12.2节），*T**到*void**（5.6节），*int*到*unsigned int*；C.6节）的匹配。
- [4] 利用用户定义转换（11.4节）的匹配。
- [5] 利用在函数声明中的省略号...（7.6节）的匹配。

如果在能找到匹配的某个最高的层次上同时发现了两个匹配，这个调用将作为存在歧义而被拒绝。这些解析规则经过了仔细的推敲，主要是为了将C和C++关于内部数值类型的精细规则（C.6节）也一并考虑在内。例如，

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
    print(c);           // 准确匹配：调用print(char)
    print(i);           // 准确匹配：调用print(int)
    print(s);           // 整数提升：调用print(int)
    print(f);           // float到double的提升：调用print(double)

    print('a');         // 准确匹配：调用print(char)
    print(49);          // 准确匹配：调用print(int)
    print(0);           // 准确匹配：调用print(int)
    print("a");         // 准确匹配：调用print(const char*)
}
```

调用`print(0)`将激活`print(int)`，是因为0是*int*。调用`print('a')`激活`print(char)`，因为'a'是*char*（4.3.1节）。将提升和转换分开的原因是我们希望能更偏向于提升，例如*char*到*int*；而不是不那么安全的转换，如*int*到*char*。

重载解析与被考虑的函数声明的顺序无关。

相对而言，重载所依赖的这组规则是比较复杂的，但程序员却很少会对哪个函数被调用感到吃惊。那么，为什么呢？请考虑一下重载的替代方式。我们经常需要对几个类型的对象执行类似操作。如果没有重载，我们就必须用几个不同的名字去定义几个函数：

```
void print_int(int);
void print_char(char);
void print_string(const char*); // C风格的字符串

void g(int i, char c, const char* p, double d)
{
    print_int(i);         // ok
    print_char(c);        // ok
```

```

    print_string(p);      // ok

    print_int(c);         // 可以吗? 调用print_int(int(c))
    print_char(i);        // 可以吗? 调用print_char(char(i))
    print_string(i);       // 错误
    print_int(d);          // 可以吗? 调用print_int(int(d))
}

```

与重载的`print()`相比,我们必须同时记住几个名字,而且要记住如何正确地使用它们。这会令人厌倦,也会挫败通用型程序设计(2.7.2节)的企图,并促使程序员去注意相对低级的类型问题。因为不存在重载函数,所有标准转换都将被用于这些函数的参数。这样做也可能引起错误。在上面例子里,这种情况的潜台词是,在这里的四个采用了“错误”参数的调用中,编译器只能捕捉到其中的一个。由此看,重载还能增加编译器拒绝不合适参数的机会。

7.4.1 重载和返回类型

重载解析中将不考虑返回类型。这样规定的理由就是要保持对重载的解析只是针对单独的运算符(11.2.1节、11.2.4节)或函数调用,与调用的环境无关。考虑

```

float sqrt(float);
double sqrt(double);

void f(double da, float fla)
{
    float fl = sqrt(da); // 调用sqrt(double)
    double d = sqrt(da); // 调用sqrt(double)
    fl = sqrt(fla);       // 调用sqrt(float)
    d = sqrt(fla);        // 调用sqrt(float)
}

```

如果把返回类型也考虑在内,我们就无法继续去孤立地去看一个`sqrt()`调用,并由此确定到底应该调用哪个函数了。

7.4.2 重载与作用域

在不同的非名字空间作用域里声明的函数不算是重载。例如,

```

void f(int);

void g()
{
    void f(double);
    f(1); // 调用f(double)
}

```

很清楚,`f(int)`应该是对`f(1)`的最好匹配,但只有`f(double)`在作用域里。对于这类情况,可以通过加入或者去除局部声明的方式去取得所需要的行为。与其他地方一样,有意识的遮蔽可以成为一种很有用的技术,但无意识的遮蔽则是产生令人吃惊情况的一个根源。如果希望重载能够跨越类作用域(15.2.2节)或名字空间作用域(8.2.9.2节),那么可以利用使用声明或者使用指令(8.2.2节)。另见8.2.6节。

7.4.3 手工的歧义性解析

对一个函数,声明的重载版本过少(或者过多)都有可能导致歧义性。例如,

```

void f1(char);
void f1(long);

void f2(char*);
void f2(int*);

void k(int i)
{
    f1(i);    // 歧义的: f1(char) 或 f1(long)
    f2(0);    // 歧义的: f2(char*) 或 f2(int*)
}

```

只要可能,在这种情况下应该做的就是将该函数的重载版本集合作为一个整体来考虑,看看对于函数的语义而言它们是否有意义。有关问题经常可以通过增加一个消解歧义性的版本而得到解决。例如,加进

```
inline void f1(int n) { f1(long(n)); }
```

就能以偏向更大类型 *long int* 的方式消解所有类似 *f1(i)* 的歧义性情况。

也可以通过增加一个显式类型转换的方式去解决某个特定调用的问题。例如,

```
f2(static_cast<int*>(0));
```

然而,这样做经常只是一种权宜之计,很快就可能遇到另一个必须处理的类似调用。

一些C++ 新手会被编译器报告出的歧义性错误弄得急躁起来。更有经验的程序员则欣赏这种错误信息,将它们看做是很有用的关于设计错误的指示器。

7.4.4 多参数的解析

有了上述重载解析规则之后,我们就可以保证:当所涉及到的不同类型在计算效率或者精度方面存在明显差异时,被调用将会是最简单的算法(函数)。例如,

```

int pow(int, int);
double pow(double, double);

complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
    int i = pow(2, 2);           // 调用pow(int, int)
    double d = pow(2.0, 2.0);    // 调用pow(double, double)
    complex z2 = pow(2, z);      // 调用pow(double, complex)
    complex z3 = pow(z, 2);      // 调用pow(complex, int)
    complex z4 = pow(z, z);      // 调用pow(complex, complex)
}

```

如果选择过程牵涉到两个或者更多的参数,那么将根据7.4节的规则为每个参数找到最佳匹配。如果有一个函数在某个参数上具有最佳的匹配,而在其他参数的匹配上都优于或者等于其他可能被调用的函数,那么它就会被调用。如果没有这样的函数,这个调用将被看做有歧义而予以拒绝。例如,

```
void g()
```

```
{
    double d = pow(2.0, 2); // 错误: pow(int(2.0), 2) 或 pow(2.0, double(2))?
}
```

这个调用具有歧义性，因为对2.0的最佳匹配是`pow(double, double)`，而对2的最佳匹配却是`pow(int, int)`。

7.5 默认参数

一个通用函数所需要的参数常常比处理简单情况时所需要的参数更多一些。特别地，那些为对象构造所用的函数（10.2.3节）通常都为灵活性而提供了一些选项。考虑一个打印整数的函数，给用户一个关于以什么为基数去打印它的选项是很合理的，但是在大部分程序里，整数都会按十进制的形式打印。例如，

```
void print(int value, int base = 10); // 默认的基是10

void f()
{
    print(31);
    print(31, 10);
    print(31, 16);
    print(31, 2);
}
```

可能产生

```
31 31 1f 11111
```

也可以通过重载得到这种默认参数的效果

```
void print(int value, int base);
inline void print(int value) { print(value, 10); }
```

当然，采用重载将使读者不容易看到原来的意图：用一个函数加上一种简写形式。

默认参数的类型将在函数声明时检查，在调用时求值。只可能对排列在最后的那些参数提供默认参数，例如，

```
int f(int, int = 0, char* = 0); // ok
int g(int = 0, int = 0, char*); // 错误
int h(int = 0, int, char* = 0); // 错误
```

请注意，在`*`和`=`之间的空格是重要的（`*=`是一个赋值运算符，6.2节）。

```
int nasty(char* = 0); // 语法错
```

在同一个作用域中随后的声明里，默认参数都不能重复或者改变。例如，

```
void f(int x = 7);
void f(int = 7); // 错误: 默认参数不能重复
void f(int = 8); // 错误: 默认参数不能改变

void g()
{
    void f(int x = 9); // 可以: 这个声明将遮蔽外层的声明
    // ...
}
```

在嵌套作用域里声明一个名字，让它去遮蔽在外层作用域里同一名字的声明，这种做法很容

易引出错误。

7.6 未确定数目的参数

对于有些函数而言，我们没办法确定在各个调用中所期望的所有参数的个数和类型。声明这种函数的方式就是在参数表的最后用省略号 (...) 结束，省略号表示“还可能有另外一些参数”。例如，

```
int printf(const char* ...);
```

这描述的是C标准库函数`printf()` (21.8节)，对它的一个调用至少必须有一个参数，一个`char*`，但是还可以有，也可以没有其他参数。例如，

```
printf("Hello, world!\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n", 2, 3, 5);
```

这样的函数必须依赖于一些编译时无法使用的信息去解释它的参数表。对于`printf()`而言，其第一个参数是个格式串，其中包含了一些特殊字符序列，这些字符使`printf()`能够正确地处理其他参数。`%s`表示“期待一个`char*`参数”，`%d`表示“期待一个`int`参数”。然而，一般来说，编译器是没有办法知道这些情况的，它不能保证所期待的参数真的就在那里，也无法保证某个参数具有合适的类型。例如，

```
#include <stdio.h>

int main()
{
    printf("My name is %s %s\n", 2);
}
```

将能通过编译，但（在最好情况下）将导致某些看起来很奇怪的输出（请试一试）。

事情很清楚，如果一个参数没有声明，编译器就没有信息去对它执行标准的类型检查和转换。在这种情况下，一个`char`或`short`将作为`int`传递，`float`将作为`double`传递^①。这些做法未必是程序员所期望的。

一个设计良好的程序至多只需要极少的几个这种参数类型没有完全刻画的函数。在大部分情况下，我们都可以利用重载函数和使用默认参数的函数来处理类型检查的问题。如果没有这些机制，在一些情况下就无法刻画参数的类型。只有在那些参数数目和参数类型都有变化的情况下才需要省略号。省略号最常见的用途是描述C库函数的界面，这些都是在C++ 提供了替代方式之前做出来的：

```
int fprintf(FILE*, const char* ...);    // 取自<stdio>
int execl(const char* ...);            // 取自UNIX头文件
```

在`<cstdarg>`里提供了一组标准的宏，专门用于在这种函数里访问未加描述的参数。现在考虑写一个出错函数，它有一个`int`参数指明错误的严重性，随后是任意个字符串。这里的想法是将各个词分别作为字符串参数传递，进而组合起这个错误信息。字符串参数列表的最后需要有一个指向`char`的空指针

① 这些都是由C语言继承来的标准提升。对于由省略号表示的参数，其实际参数在传递之前总执行这些提升（如果它们属于需要提升的类型），将提升之后的值传递给有关的函数。——译者注

```

extern void error(int ...);
extern char* itoa(int, char[]); // 见6.6[17]
const char* Null_cp = 0;

int main(int argc, char* argv[])
{
    switch (argc) {
        case 1:
            error(0, argv[0], Null_cp);
            break;

        case 2:
            error(0, argv[0], argv[1], Null_cp);
            break;

        default:
            char buffer[8];
            error(1, argv[0], "with", itoa(argc-1, buffer), "arguments", Null_cp);
    }
    // ...
}

```

函数`itoa()`返回它的整数参数的字符串表示。

请注意，用整数0作为结束符可能产生不可移植问题：在某些实现中，整数0和空指针的表示形式可能不同。这一情况也说明，一旦使用省略号抑制了类型检查，程序员就必须直接面对一些难以琢磨的额外工作。

出错函数可以像下面这样定义：

```

void error(int severity ...) // "severity" (严重性) 后跟空指针结束的char* 列表
{
    va_list ap;
    va_start(ap, severity); // arg开始
    for (;;) {
        char* p = va_arg(ap, char*);
        if (p == 0) break;
        cerr << p << ' ';
    }
    va_end(ap); // arg清理
    cerr << '\n';
    if (severity) exit(severity);
}

```

首先，通过调用`va_start()`定义并初始化一个`va_list`。宏`va_start`以一个`va_list`的名字和函数的最后一个有名形式参数的名字作为参数。宏`va_arg()`用于按顺序提取出各个无名参数。在每次调用`va_arg()`时，程序员都必须提供一个类型，`va_arg()`假定这就是被传递的实际参数的类型，但一般说它并没有办法去保证这一点。从一个使用过`va_start()`的函数中退出之前，必须调用一次`va_end()`。这是因为`va_start()`可能以某种方式修改了堆栈，这种修改可能导致返回无法完成，`va_end()`能将有关的修改复原。

7.7 指向函数的指针

对于一个函数只能做两件事：调用它，或者取得它的地址。通过取一个函数的地址而得

到的指针，可以在后面用于调用这个函数。例如，

```
void error(string s) { /* ... */ }

void (*efct)(string);    // 指向函数的指针

void f()
{
    efct = &error;        // efct指向error
    efct("error");        // 通过efct调用error
}
```

编译器知道`efct`是一个指针，并会去调用被指的函数。这也就是说，可以不写从指针得到函数的间接运算`*`。与此类似，取得函数地址的`&`也可以不写：

```
void (*f1)(string) = &error;    // ok
void (*f2)(string) = error;    // 也可以；与 &error意思一样

void g()
{
    f1("Vasa");                // ok
    (*f1)("Mary Rose");        // 也可以
}
```

在指向函数的指针的声明中也需要给出参数类型，就像函数声明一样。在指针赋值时，完整的函数类型必须完全匹配。例如，

```
void (*pf)(string);    // 指向void(string)
void f1(string);        // void(string)
int f2(string);         // int(string)
void f3(int*);          // void(int*)

void f()
{
    pf = &f1;            // ok
    pf = &f2;            // 错误：返回类型不对
    pf = &f3;            // 错误：参数类型不对

    pf("Hera");          // ok
    pf(1);               // 错误：参数类型不对

    int i = pf("Zeus");  // 错误：void赋值给int
}
```

无论直接调用函数或通过某个指针去调用函数，有关参数传递的规则都完全相同。

人们常常为了方便而为指向函数的指针类型定义一个名字，这样可以避免到处去写意义不太明显的语法形式。下面是来自UNIX系统头文件的一个例子：

```
typedef void (*SIG_TYP)(int);    // 取自<signal.h>
typedef void (*SIG_ARG_TYP)(int);
SIG_TYP signal(int, SIG_ARG_TYP);
```

指向函数的指针的数组常常很有用。例如，我的基于鼠标的编辑器里的菜单系统，就是利用指向函数的指针的数组实现的，这些函数表示各种各样的操作。这个系统的细节不可能在这里描述，但下面是其中的基本想法：

```
typedef void (*PF)();

PF edit_ops[] = {            // 编辑操作
```

```

        &cut, &paste, &copy, &search
    };

    PF file_ops[] = {          // 文件管理
        &open, &append, &close, &write
    };

```

然后我们就可以定义并初始化一些指针，由它们去控制各种操作，通过关联于鼠标键的菜单去选择那些操作：

```

    PF* button2 = edit_ops;
    PF* button3 = file_ops;

```

在一个完整的实现里，定义一个菜单项需要提供更多的信息。例如，必须在某个地方保存有关需要显示的字符串的一个描述。随着系统的使用，鼠标键的意义也可能随环境而频繁变化，这种变化就可以（部分地）通过修改按键所对应的指针来实现。当用户选择一个菜单项时，例如按键2的项目3，就会执行相关的操作：

```

    button2[2](); // 调用按键2的第3个函数

```

要理解指向函数的指针的表达能力，一种方式就是试着去写这种代码而不用函数指针——也不用它们的更具良好行为的兄弟：虚函数（12.2.6节）。通过把新函数插入运算符表等方式，就可以在运行中修改这种菜单。在运行中构造出新菜单也同样非常容易。

指向函数的指针可以用于提供一种简单形式的多态性例程，即那种可以应用于许多不同类型的对象的例程：

```

typedef int (*CFT) (const void*, const void*);

void ssort(void* base, size_t n, size_t sz, CFT cmp)
/*
    对向量base的n个元素按照递增顺序排序，
    用由"cmp"所指的函数做比较，
    元素的大小是"sz"。

    Shell排序（Knuth, Vol.3, Pg84）
*/
{
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap) {
                char* b = static_cast<char*>(base); // 必须强制
                char* pj = b+j*sz;                  // &base[j]
                char* pjg = b+(j+gap)*sz;            // &base[j+gap]

                if (cmp(pjg, pj)<0) {                  // 交换base[j] 与base[j+gap]:
                    for (int k=0; k<sz; k++) {
                        char temp = pj[k];
                        pj[k] = pjg[k];
                        pjg[k] = temp;
                    }
                }
            }
    }
}

```

`ssort()` 例程并不知道被排序的对象的类型，它只知道元素的个数（数组大小），每个元素的大

小, 以及应该去调用以完成比较的函数。这里有意将`ssort()`的类型选的与C标准库排序例程`qsort()`完全一样。实际程序可以使用`qsort()`, C++ 标准库算法`sort` (18.7.1节), 或者其他特殊的排序例程。这种风格的代码在C里很常见, 但它不是在C++ 里表述这个算法的最优美的方式 (13.3节、13.5.2节)。

这个排序函数可用于对下面这样的表格的排序:

```
struct User {
    char* name;
    char* id;
    int dept;
};

User heads[] = {
    "Ritchie D.M.", "dmr", 11271,
    "Sethi R.", "ravi", 11272,
    "Szymanski T.G.", "tgs", 11273,
    "Schryer N.L.", "nls", 11274,
    "Schryer N.L.", "nls", 11275,
    "Kernighan B.W.", "bwk", 11276
};

void print_id(User* v, int n)
{
    for (int i=0; i<n; i++)
        cout << v[i].name << '\n' << v[i].id << '\n' << v[i].dept << '\n';
}
```

为能完成排序, 我们首先需要定义一个适当的比较函数。这种比较函数应该在其第一个参数小于第二个参数时返回负值, 如果它们相等就返回0, 否则就返回正值:

```
int cmp1(const void* p, const void* q) // 比较名字串
{
    return strcmp(static_cast<const User*>(p)->name, static_cast<const User*>(q)->name);
}

int cmp2(const void* p, const void* q) // 比较部门编号
{
    return static_cast<const User*>(p)->dept - static_cast<const User*>(q)->dept;
}
```

下面程序完成排序和打印:

```
int main()
{
    cout << "Heads in alphabetical order:\n";
    ssort(heads, 6, sizeof(User), cmp1);
    print_id(heads, 6);
    cout << '\n';

    cout << "Heads in order of department number:\n";
    ssort(heads, 6, sizeof(User), cmp2);
    print_id(heads, 6);
}
```

你可以通过赋值或者初始化指向函数的指针的方式, 取得一个重载函数的地址。在这种情况下, 从一组重载函数中选择的工作是通过目标指针的类型实现的。例如,

```
void f(int);
```

```
int f(char);

void (*pf1)(int) = &f;    // void f(int)
int (*pf2)(char) = &f;    // int f(char)
void (*pf3)(char) = &f;    // 错误: 没有void f(char)
```

要通过指向函数的指针调用的函数，其参数类型和返回值类型都必须与指针的要求完全一致，在用函数对指针赋值或初始化时，没有隐含的参数或者返回值类型转换。这意味着

```
int cmp3(const mytype*, const mytype*);
```

不是`ssort()`的合适参数。究其原因，接受`cmp3`作为`ssort`的参数将会违反有关的保证：`cmp3`调用时参数的类型是`mytype*`（9.2.5节）。

7.8 宏

宏在C语言里极其重要，而在C++里用得就少多了。关于宏的第一规则是：绝不应该去使用它，除非你不得不这样做。几乎每个宏都表明了程序设计语言里，或者程序里，或者程序员的一个缺陷，因为它将在编译器看到程序的正文之前去重新摆布这些正文。宏也是许多程序设计工具的主要麻烦。所以，如果你使用了宏，你就应该准备着只能从各种工具（如排错系统、交叉引用工具、轮廓程序等）得到较少的服务。如果你必须使用宏，那么请仔细阅读你所用的C++预处理器实现的手册，而且不用过于自作聪明。还要警告读者，应该按照习惯，在为宏命名时使用许多大写字母。宏的语法在A.11节给出。

简单的宏定义就像下面这样：

```
#define NAME rest of line
```

当`NAME`作为一个单词被遇到时，它就会被用字符序列`rest of line`（该行剩下的部分）取代。例如，

```
named = NAME
```

将被展开为

```
named = rest of line
```

也可以定义带参数的宏，例如，

```
#define MAC(x,y) argument1: x argument2: y
```

在使用`MAC`时必须提供两个字符串^①，它们将在`MAC()`被展开时用于取代`x`和`y`，例如，

```
expanded = MAC(foo bar, yuk yuk)
```

将展开成

```
expanded = argument1: foo bar argument2: yuk yuk
```

宏名字不能重载，而且宏预处理器不能处理递归调用：

```
#define PRINT(a,b) cout<<(a)<<(b)
#define PRINT(a,b,c) cout<<(a)<<(b)<<(c) /* 麻烦? : 重复定义, 不允许重载 */
#define FAC(n) (n>1)?n*FAC(n-1):1 /* 麻烦: 递归的宏 */
```

① 请注意，本小节中所说的“字符串”指的就是写在程序正文里的一段字符。与C风格的字符串或者C++标准库定义的`string`完全无关（那些都是程序中的对象）。——译者注

宏对字符串进行操作，它对C++的语法知之甚少，根本不知道C++的类型和作用域规则。编译器能看到的只是宏展开后的形式，所以在宏中的错误是在宏被展开之后报告的，而不是在它定义时，这可能导致非常难以理解的错误信息。

下面是一些可能有用的宏

```
#define CASE break; case
#define FOREVER for(;;)
```

下面是一些完全没有必要的宏：

```
#define PI 3.141593
#define BEGIN {
#define END }
```

下面是一些很危险的宏：

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
```

想知道它们为什么危险，请试着展开这个程序片段：

```
int xx = 0;      // 全局计数器

void f()
{
    int xx = 0;      // 局部变量
    int y = SQUARE(xx+2); // y=xx+2*xx+2; 即y=xx+(2*xx)+2
    INCR_xx;        // 增加局部的xx
}
```

如果你要使用宏，在引用全局名字时一定要使用作用域解析运算符 :: (4.9.4节)，并在所有可能的地方将出现的宏参数都用括号围起来。例如，

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

如果你写的宏足够复杂，需要写注释时，采用 /* */ 形式的注释是比较明智的，因为有时会有不懂C++的 // 注释的C预处理程序被用做C++工具的一部分。例如，

```
#define M2(a) something(a) /* 细心的注释*/
```

通过利用宏，你可以设计出自己的私有语言。即使与C++相比你更偏爱自己的这种“增强的语言”，但对于大部分C++程序员而言，它也是不可理解的。进一步说，C预处理器是一种很简单的宏处理器。当你试图去做某些不那么简单的事情时，你多半会发现不能做好，或者做起来有不必要的困难。*const*、*inline*、*template*、*enum*和*namespace*机制等都是为了用做预处理器结构的许多传统使用方式的替代品。例如，

```
const int answer = 42;
template<class T> inline T min(T a, T b) { return (a<b)?a:b; }
```

在写宏的时候，需要为某些东西提供新名字也是很常见的事情。通过 ## 宏运算符可以拼接起两个串，构造出一个新串。例如，

```
#define NAME2(a,b) a##b
int NAME2(hack, cah)();
```

将产生

```
int hackcah();
```

提供给编译器去读。

指令

```
#undef X
```

保证不再有称为 X 的有定义的宏——无论在此指令之前有还是没有。这种东西可以用于防止某些不想要的宏。然而，要想知道 X 对一片代码的作用是否如自己所设想的那样，那可总是不是一件容易的事情。

7.8.1 条件编译

有一种宏的使用几乎不可能避免。指令 `#ifdef identifier` 将条件性地导致随后的输入被忽略，直到遇到一个 `#endif` 指令。例如，

```
int f(int a
#ifdef arg_two
, int b
#endif
);
```

产生

```
int f(int a
);
```

给编译器去看，除非名为 `arg_two` 的宏已经用 `#define` 定义。这种例子将会把工具弄糊涂，而程序员通常总认为它们的行为是合理的。

`#ifdef` 的大部分使用没有这么古怪。有节制地使用 `#ifdef` 不会有什么害处。另见 9.3.3 节。用于控制 `#ifdef` 的名字应该仔细选择，使之不会与正常的标识符相冲突。例如，

```
struct Call_info {
    Node* arg_one;
    Node* arg_two;
    // ...
};
```

这样看起来很清白的代码也可能产生混乱，假如某人写了：

```
#define arg_two x
```

不幸的是，普通的、无法避免的头文件里包含着许多危险且毫无必要的宏。

7.9 忠告

- [1] 质疑那些非 `const` 的引用参数；如果你想要一个函数去修改其参数，请使用指针或者返回值；5.5 节。
- [2] 当你需要尽可能减少参数复制时，应该使用 `const` 引用参数；5.5 节。
- [3] 广泛而一致地使用 `const`；7.2 节。
- [4] 避免宏；7.8 节。

- [5] 避免不确定数目的参数；7.6节。
- [6] 不要返回局部变量的指针或者引用；7.3节。
- [7] 当一些函数对不同的类型执行概念上相同的工作时，请使用重载；7.4节。
- [8] 在各种整数上重载时，通过提供函数去消除常见的歧义性；7.4.3节。
- [9] 在考虑使用指向函数的指针时，请考虑虚函数（2.5.5节）或模板（2.7.2节）是不是更好的选择；7.7节。
- [10] 如果你必须使用宏，请使用带有许多大写字母的丑陋的名字；7.8节。

7.10 练习

1. (*1) 写出下面声明：一个函数，它以指向字符的指针和对整数的引用为参数，不返回值；一个指向这个函数的指针；一个以这种指针为参数的函数；以及一个返回这种指针的函数。写出一个函数的定义，它以一个这样的指针作为参数，并返回其参数作为返回值。提示：使用***typedef***。

2. (*1) 下面的代码是什么意思？它会有什么用处？

```
typedef int (&rfii) (int, int);
```

3. (*1.5) 写出一个类似“Hello, world!”的函数，它以一个名字作为命令行参数，并写出“Hello, *name*”（其中*name*是实际的命令行参数）。修改这个程序，使它能以一系列名字作为参数，并对每个名字分别说hello。

4. (*1.5) 写一个程序，它读入任意多个由命令行参数提供名字的文件，并将它们一个接着一个写入***cout***。因为这个程序拼接起它的输入去产生输出，你可以称它为***cat***。

5. (*2) 将一个小的C程序转换为C++ 程序。修改头文件，声明所有的函数调用，并声明所有的参数类型。只要可能，就把***#define***换成***enum***、***const***或者***inline***。从***.c***文件里删除所有的***extern***声明。如果有必要的话，将所有的函数定义转换为C++ 的函数定义语法。将所有对***malloc()***和***free()***的调用替换为***new***和***delete***。删除不必要的强制。

6. (*2) 用更高效的排序算法实现***ssort()***（7.7节） 提示：***qsort()***。

7. (*2.5) 考虑：

```
struct Tnode {
    string word;
    int count;
    Tnode* left;
    Tnode* right;
};
```

写一个函数向***Tnode***的树中插入新的单词。写一个函数将***Tnode***的树打印出来。写一个函数将***Tnode***的树按照单词的字典顺序打印出来。修改***Tnode***，使得它只存储一个到任意长的单词的指针，该单词存储在一个由***new***分配的字符数组里。修改上述函数，使它们使用新的***Tnode***。

8. (*2.5) 写一个函数求二维数组的逆。提示：C.7节。

9. (*2) 写一个加密程序，它从***cin***读入，并将编码后的字符序列写到***cout***。你可以采用如下的简单加密模式：字符***c***的加密形式是 **$c \wedge key[i]$** ，其中***key***是通过命令行参数提供的一个字符串。这个程序以循环的方式使用***key***中的字符，直到读完全部输入。用同一个***key***重新加密

编码后的正文就能得到原来的正文。如果不提供`key`（即提供空字符串），则不做加密。

10. (*3.5) 写一个程序来帮助在不知道`key`的情况下解密采用7.10[9]的方式加密的消息。提示：参看David Kahn, 《The Codebreaker》, Macmillan, 1967, New York, 207~213页。
11. (*3) 写一个`error`函数，它取一个`printf`风格的包含`%s`、`%c`和`%d`指示符的格式串，以及任意多个其他参数。请不要使用`printf()`。如果你不知道`%s`、`%c`和`%d`的意思，请参看21.8节。使用`<stdarg.h>`。
12. (*1) 你怎样为指向函数的指针类型选择`typedef`所定义的名字？
13. (*2) 看一些程序，以便取得对实际程序中名字使用的各种风格的一些认识。大写字母如何用？下划线符如何用？什么时候使用如`i`和`x`这样的短名字？
14. (*1) 下面这些宏定义各有什么毛病？

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)*fac((a)-1)
```

15. (*3) 写一个宏处理器，它能够定义和展开简单的宏（像C预处理器那样）。它从`cin`读并向`cout`写。开始时请不要去处理带参数的宏。提示：桌面计算器（6.1节）包含一个符号表和一个词法分析器，你可以修改它们。
16. (*2) 实现7.5节的`print()`。
17. (*2) 给6.1节的桌面计算器增加函数，例如`sqrt()`、`log()`和`sin()`。提示：预先定义一些名字，通过一个指向函数的指针的数组调用这些函数。不要忘记对于函数调用检查参数的类型。
18. (*1) 写一个不用递归的阶乘函数。参见11.14[6]。
19. (*2) 写一个函数，实现为5.9[13]定义的`Date`加上一天、一个月、一年的功能。写一个函数对于所给的`Date`给出对应的星期几。写一个函数给出参数`Date`之后第一个星期一对应的`Date`。

第8章 名字空间和异常

那年是787年！

公元？

——Monty Python

任何规则都不可能如此一般，
以至不能容许任何例外（异常）。

——Robert Burton

模块化、界面和异常——名字空间——*using*——*using namespace*——避免名字冲突
——名字查找——名字空间组合——名字空间别名——名字空间和C代码——异常——
*throw*和*catch*——异常和程序结构——忠告——练习

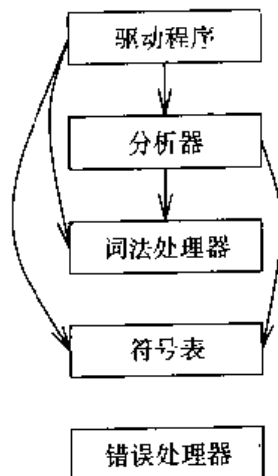
8.1 模块化和界面

任何实际程序都是由一些部分组成的。例如，甚至最简单的“Hello, world!”程序也涉及到至少两部分：用户代码要求将***Hello, world!***打印出来，I/O系统完成打印工作。

考虑6.1节的桌面计算器，可以将它看做是由5个部分组成：

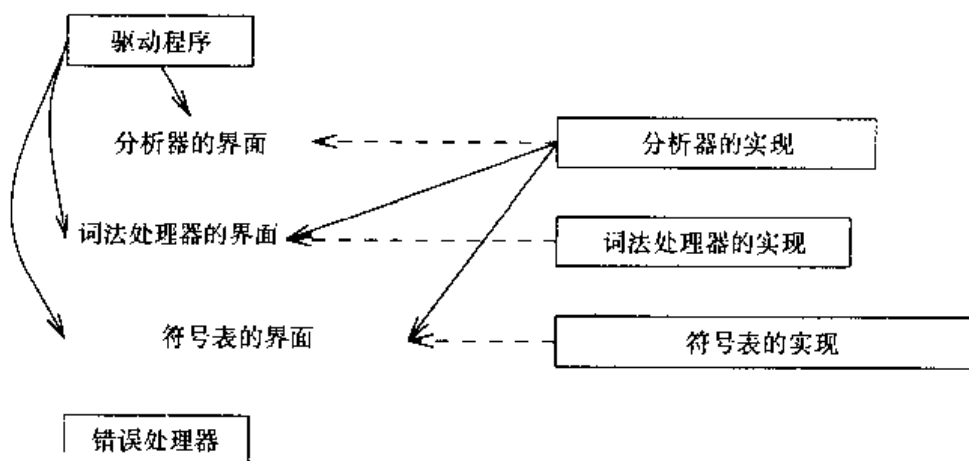
- [1] 一个分析器，完成语法分析。
- [2] 一个词法处理器，由字符组合出单词。
- [3] 一个符号表，保存字符串和值的对偶。
- [4] 一个驱动程序*main()*。
- [5] 一个错误处理器。

有关情况可以用如下图形表示：



这里的箭头表示“使用”。为了简化这个图，我没有表示出每个部分都依赖于错误处理的这一事实。实际上，可以将这个计算器设想为由三部分组成，加上驱动程序和错误处理器只是为了完全。

当一个模块使用另一个模块时，它并不需要知道有关被用模块的所有东西。理想的情况是，一个模块的大部分细节都不为其使用者所知。为此，我们就需要将一个模块和它的界面区分开来。举例来说，分析器直接依赖于词法处理器的界面（仅此而已），并不依赖于整个词法处理器。该词法处理器不过是实现了它的界面所声言的那些服务。这些情况可以用下面图形表示：



虚线箭头表示实现。我认为这才是该程序的实际结构，而我们作为程序员的工作就是在代码里忠实地表达这些东西。如果真的做到了这些，那么结果代码就会是简单的、高效的、易理解的、可维护的，如此等等，因为它是我们的基本设计的直接反映。

下面各节将展示怎样将桌面计算器的逻辑结构弄得更清晰些，在9.3节将显示我们可能怎样物理地将有关的程序正文组织起来，以获得这样做的一些优点。这个计算器是一个很小的程序，所以，在“现实生活中”我不会费心地去对它使用名字空间和分别编译（2.4.1节、9.1节），至少不会做到这里所做的程度。这里这样做就是为了展示用于更大型的程序的有用技术，又不至于使我们淹没在代码里。在实际程序里，每个由单独名字空间表示的“模块”常常会包含成百个函数、类、模板等。

为了阐释不同的技术和语言特征，我将分阶段开发计算器的模块化。在“现实生活中”，一个程序不大可能会经过这么多阶段而成长起来。一个有经验的程序员可能从一开始就捡起某个“大致正确”的设计。当然，随着一个程序历经多年的演化，出现剧烈的结构改变也不是很罕见的事情。

错误处理将遍及程序的整个结构。在将一个程序分解为模块时，或者（相反地）从一些模块组合出程序时，我们都必须特别注意将由错误处理造成的模块之间的相互依赖减到最小。C++ 提供了异常机制，用于降低检查、报告错误和处理错误之间的联系程度。因此，在讨论了如何将模块表示为名字空间（8.2节）之后，我们将阐释怎样利用异常进一步改善模块性（8.3节）。

现存的有关模块的概念比本章和下一章将要讨论的东西多得多。例如，我们还可能使用并发执行的和相互通信的进程来表示模块性的一些重要方面。类似地，独立地址空间的使用以及不同地址空间之间的通信也是很重要的论题，这些在这里都没有讨论。我认为有关模块

的这些概念在很大程度上是互相独立、彼此正交的。有趣的是，在每种情况下，将系统划分为模块都很容易，最困难的是提供跨过模块边界的安全、方便而有效的通信。

8.2 名字空间

名字空间是一种描述逻辑分组的机制。也就是说，如果有一些声明按照某种准则在逻辑上属于同一个集团，就可以将它们放入同一个名字空间，以表明这个事实。例如，在桌面计算器（6.1.1节）中有关分析器的声明就可以放入一个名字空间**Parser**：

```
namespace Parser {
    double expr(bool);
    double prim(bool get) { /* ... */ }
    double term(bool get) { /* ... */ }
    double expr(bool get) { /* ... */ }
}
```

函数**expr()**必须首先声明而后再定义，以便能打开在6.1.1节所说的依赖性循环。

桌面计算器的输入部分也可以放入自己的名字空间：

```
namespace Lexer {
    enum Token_value {
        NAME,          NUMBER,          END,
        PLUS='+',      MINUS='-',      MUL='*',      DIV='/',
        PRINT=';',      ASSIGN='=',      LP='(',      RP=')'
    };

    Token_value curr_tok;
    double number_value;
    string string_value;

    Token_value get_token() { /* ... */ }
}
```

这样使用名字空间，将词法处理器和分析器为用户提供了些什么样的情况表现得相当明显。然而，如果我将函数的源代码也包括进来，这个结构就会变模糊了。在实际大小的名字空间里，如果将各个函数体也包括在声明中，你将经常需要翻过许多页或者许多屏信息，去寻找模块所提供的服务，即寻找它的界面。

也存在另一种方式可以代替基于分开描述的界面，那就是提供一种从包含着实现细节的模块里抽取界面信息的工具。我并不认为那是一种好办法。描述模块的界面是一种最基本的设计活动（23.4.3.4节），同一个模块可以为不同的用户提供不同的界面，而且，界面的设计通常也是远在实现细节变得更具体之前进行的。

这里是**Parser**的一个版本，其中将界面与实现分离了：

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
}

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

请注意，作为将实现与界面分离的结果，现在的每个函数只有一个声明和一个定义。用户看

到的只是界面里所包含的声明，而实现——在这里就是函数体——将被放在“其他的什么地方”，用户不需要去看。

如上所示，成员可以在名字空间的定义里声明，而后采用`namespace-name::member-name`的形式去定义。

一个名字空间的成员必须采用如下的记法形式引入：

```
namespace namespace-name {
    // 声明和定义
}
```

我们不能在名字空间定义之外用加限定的语法形式为名字空间引进新成员。例如，

```
void Parser::logical(bool);    // 错误：Parser里没有logical()
```

这里的想法就是为了更容易找到一个名字空间里的所有名字，也能捕捉到例如拼写或者类型不匹配一类的错误。例如，

```
double Parser::trem(bool);    // 错误：Parser里没有term()
double Parser::prim(int);     // 错误：Parser::prim() 要求一个bool参数
```

一个名字空间也是一个作用域。这样，“名字空间”就是一种最基本的相对简单的概念。一个程序越大，通过名字空间去描述其中逻辑上独立的各个部分也就越重要。常规的局部作用域、全局作用域和类也都是名字空间（C.10.3节）。

理想情况是，程序里的每个实体都属于某个可以识别的逻辑单位（“模块”）。所以，在理想情况下，一个非平凡的程序里的每个声明都应该位于某个名字空间里，以此指明它在程序中所扮演的逻辑角色。例外的是`main()`，它必须是特殊的，以便运行时环境能够特殊对待它（8.3.3节）。

8.2.1 带限定词的名字

因为名字空间是作用域，所以普通的作用域规则也对名字空间成立。因此，如果一个名字先前已在本名字空间里或者其外围作用域里声明过，它就可以直接使用了，不必再进一步为它操心。也可以使用来自另一个名字空间的名字，但需要用该名字所属的名字空间作为限定词。例如，

```
double Parser::term(bool get)    // 请注意Parser:: 限定词
{
    double left = prim(get);      // 不需要限定词
    for (;;)
        switch (Lexer::curr_tok) { // 注意Lexer:: 限定词
            case Lexer::MUL:        // 注意Lexer:: 限定词
                left *= prim(true); // 不需要限定词
            // ...
        }
    // ...
}
```

第一行的`Parser::`限定词是必需的，用以说明这个`term()`就是在`Parser`里所声明的那一个，而不是别的什么不相干的全局函数。由于`term()`是`Parser`的成员，因此就不需要再为`prim()`使用限定词了。但是，如果没写`Lexer`限定词，`curr_tok`将会被认为是没有声明的，因为名字空

间`Lexer`的成员在名字空间`Parser`里都不处在作用域之中。

8.2.2 使用声明

当某个名字在它自己的名字空间之外频繁使用时，在反复写它的时候都要加上有关名字空间作为限定词，这样做也会变成很令人厌烦的事情。考虑

```
double Parser::prim(bool get)    // 处理初等项
{
    if (get) Lexer::get_token();

    switch (Lexer::curr_tok) {
    case Lexer::NUMBER:          // 浮点常量
        Lexer::get_token();
        return Lexer::number_value;

    case Lexer::NAME:
        { double& v = table[Lexer::string_value];
          if (Lexer::get_token() == Lexer::ASSIGN) v = expr(true);
          return v;
        }

    case Lexer::MINUS:           // 一元减
        return -prim(true);

    case Lexer::LP:
        { double e = expr(true);
          if (Lexer::curr_tok != Lexer::RP) return Error::error(") expected");
          Lexer::get_token();    // 吃掉 ')'
          return e;
        }

    case Lexer::END:
        return 1;

    default:
        return Error::error("primary expected");
    }
}
```

反复写限定词`Lexer`很讨厌，也会分散人的注意力。这种多余的东西可以通过一个使用声明而清除掉，只需要在某个地方说明，在这个作用域里所用的`get_token`就是`Lexer`的`get_token`。例如，

```
double Parser::prim(bool get)    // 处理初等项
{
    using Lexer::get_token;    // 使用Lexer的get_token
    using Lexer::curr_tok;    // 使用Lexer的curr tok
    using Error::error;        // 使用Error的error

    if (get) get_token();

    switch (curr_tok) {
    case Lexer::NUMBER:          // 浮点常量
        get_token();
        return Lexer::number_value;

    case Lexer::NAME:
        { double& v = table[Lexer::string_value];
          if (get_token() == Lexer::ASSIGN) v = expr(true);
          return v;
        }

    case Lexer::MINUS:           // 一元减
```

```

        return -prim(true);
    case Lexer::LP:
    {
        double e = expr(true);
        if (curr_tok != Lexer::RP) return error(") expected");
        get_token(); // 吃掉 ')'
        return e;
    }
    case Lexer::END:
        return 1;
    default:
        return error("primary expected");
    }
}

```

使用声明将引进局部的同义词。

将同义词尽可能地保持在局部中以避免混乱，这是一种很好的想法。由于所有分析函数都使用差不多同样的一集来自其他模块的名字，我们也可以把有关的使用声明放在**Parser**名字空间的定义里：

```

namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);

    using Lexer::get_token; // 使用Lexer的get_token
    using Lexer::curr_tok;  // 使用Lexer的curr_tok
    using Error::error;     // 使用Error的error
}

```

这就使我们能将**Parser**函数简化到几乎等同于原来的版本了：

```

double Parser::term(bool get) // 乘和除
{
    double left = prim(get);
    for (;;)
        switch (curr_tok) {
            case Lexer::MUL:
                left *= prim(true);
                break;

            case Lexer::DIV:
                if (double d = prim(true)) {
                    left /= d;
                    break;
                }
                return error("divide by 0");
            default:
                return left;
        }
}

```

也可以将各个单词的名字引进**Parser**名字空间。然而我还是采用带显式限定词的形式把它们放在那里，以帮助我们记住**Parser**依赖于**Lexer**。

8.2.3 使用指令

如果我们的目的就是**Parser**函数简化到正好是原来那个版本，那又该怎么做呢？对于一

个大程序而言，如果需要将它从以前模块化程度较低版本转为使用名字空间的话，这也是一个很合理的想法。

一个使用指令能把来自一个名字空间的所有名字都变成可用的，几乎像它们原来就声明在其名字空间之外一样（8.2.8节）。例如，

```
namespace Parser {
    double prim(bool);
    double term(bool);
    double expr(bool);
    using namespace Lexer; // 使来自Lexer的所有名字都可以用
    using namespace Error; // 使来自Error的所有名字都可以用
}
```

这就使我们可以完全按照原来的方式写*Parser*的函数（6.1.1节）：

```
double Parser::term(bool get) // 乘和除
{
    double left = prim(get);
    for (;;)
        switch (curr_tok) { // Lexer的curr_tok
            case MUL: // Lexer的MUL
                left *= prim(true);
                break;
            case DIV: // Lexer的DIV
                if (double d = prim(true)) {
                    left /= d;
                    break;
                }
                return error("divide by 0"); // Error的error
            default:
                return left;
        }
}
```

全局性的使用指令是一种完成转变的工具（8.2.9节），在其他方面最好避免使用。在一个名字空间里的使用指令是一种名字空间的组合工具（8.2.8节）。在一个函数里（也只在这种地方），可以安全地将使用指令作为一种方便的记法方式（8.3.3.1节）。

8.2.4 多重界面

应该清楚，我们为*Parser*演化出来的名字空间定义并不是*Parser*提供给它的用户的界面。相反，它是为能方便地写出各个函数所需要的一组声明。*Parser*提供给它的用户的界面远比这简单得多：

```
namespace Parser {
    double expr(bool);
}
```

幸运的是，*Parser*的这两个名字空间定义可以共存，这就使各种定义可以被用到最合适的地方。我们看到名字空间*Parser*被用于提供两种东西：

- [1] 实现分析器的所有函数的一个公共环境。
- [2] 分析器提供给它的用户的一个外部界面。

这样，驱动程序`main()`就应该只看到：

```
namespace Parser {           // 给用户的界面
    double expr(bool);
}
```

而实现`Parser`的那些函数则应该看到我们前面所确定的，对表述这些函数的共享环境而言最合适的界面，即

```
namespace Parser {           // 给实现的界面
    double prim(bool);
    double term(bool);
    double expr(bool);

    using Lexer::get_token; // 使用Lexer的get_token
    using Lexer::curr_tok;  // 使用Lexer的curr_tok
    using Error::error;     // 使用Error的error
}
```

或用图形表示



箭头表示“依赖于界面所提供的”关系。

`Parser'` 是提供给用户的最小界面。名字`Parser'` 并不是一个C++ 标识符，它只是有意选来说明这个界面在程序里并没有单独的名字。缺少另一个独立的名字并不会引起混乱，因为程序员很自然地会为不同界面发明不同的而又明显的名字，也因为程序的物理布局（9.3.2节）能很自然地提供一些独立的（文件）名字。

为实现所提供的界面比为用户提供的界面更大一些。如果这个界面针对的是现实程序里的一个现实规模的模块，那么它通常也会比用户能够看到的界面变化得更频繁些。将模块的用户（在这里是`main()`使用`Parser`）与这些变化隔离开是非常重要的。

我们并不需要两个互相独立的名字空间来描述这两个不同的界面。如果真需要的话我们也能做得到。设计界面是最基本的设计活动之一，而且是一种可以获得或者丧失重要利益的活动。因此，我们很值得去考虑到底想达到什么结果，并讨论若干不同的选择。

请记住，这里所介绍的解决方案是我们考虑过的各种解中最简单的，通常也是最好的。它的主要弱点是两个界面没有采用不同的名字，编译器不一定有足够的信息去检查两个名字空间定义的一致性。当然，即使编译器未必总有机会去检查这种一致性，它通常还是会这样做。进一步说，连接系统将能捕捉到编译器遗漏的大部分错误。

这里给出的解也是我将要用于讨论物理模块化（9.3节）的解，也是我想推荐的没有进一步逻辑约束的解（8.2.7节）。

8.2.4.1 界面设计的各种选择

界面的作用就是尽可能减小程序不同部分之间的相互依赖。最小的界面将会使程序易于理解，有很好的数据隐蔽性质，容易修改，也编译得更快。

在考虑依赖性的时候，一件很重要的事就是记住编译器和程序员都倾向于取一种对他们

最朴素的认识：“如果一个定义在位置X处于作用域之中，那么在位置X写出的所有东西都将依赖于这个定义所说的所有东西”。在典型情况下，问题实际上不会那么糟，因为大部分定义与大部分代码无关。看看我们前面使用的定义，考虑

```
namespace Parser {           // 给实现的界面
    // ...
    double expr(bool);
    // ...
}

int main()
{
    // ...
    Parser::expr(false);
    // ...
}
```

函数`main()`只依赖于`Parser::expr()`，但这需要耗费时间、脑力、计算等去弄清楚。因此，对于真实规模的程序，人和编译系统常常会按照最安全的方式去做，在那些可能有依赖的地方，就假定它确实有。通常这也是完全合理的方式。

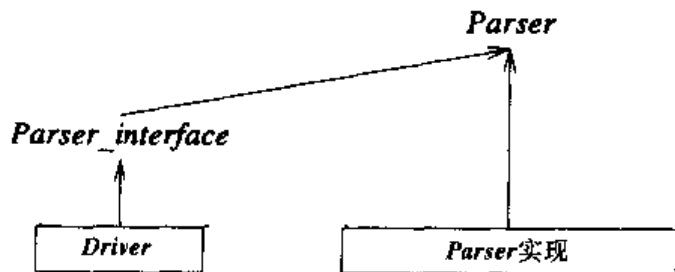
在这种情况下，我们的目标就应该是去描述自己的程序，将这种具有潜在依赖性的集合缩小到实际依赖性的集合。

我们首先试最明显的东西，用我们已有的实现界面为`Parser`定义一个用户界面：

```
namespace Parser {           // 给实现的界面
    // ...
    double expr(bool);
    // ...
}

namespace Parser_interface { // 给用户的界面
    using Parser::expr;
}
```

事情很清楚，`Parser_interface`的用户仅仅依赖于`Parser::expr()`，而且是间接的。然而，对于依赖图的粗略观察给我们的是：



现在，看起来`Driver`还是很容易受到`Parser`界面修改的伤害，而原本是希望它能与之隔离。甚至依赖关系的这种外表形式也是我们所不希望的，所以我们应显式地限制`Parser_interface`对`Parser`的依赖，使得在定义`Parser_interface`的位置，只有分析器实现界面中有关的部分（它以前被称做`Parser'`）在作用域中

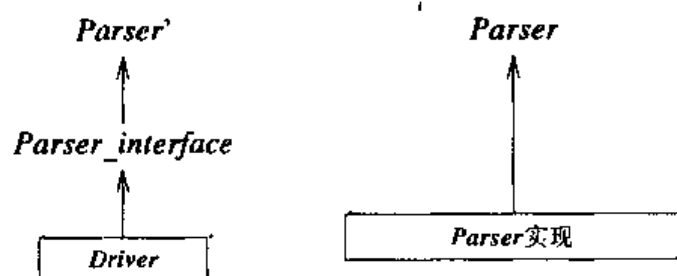
```
namespace Parser {           // 给用户的界面
    double expr(bool);
```

```

}
namespace Parser_interface { // 给用户的独立命名的界面
    using Parser::expr;
}

```

或者用图形



为了保证`Parser`和`Parser'`的一致性, 我们还是需要依赖于编译系统的整体, 而不能只靠在单独的编译单位上工作的编译器。这种解法与8.2.4节的不同之处只是另有了一个名字空间`Parser_interface`。如果我们希望的话, 也可以给`Parser_interface`一个自己的`expr()`函数:

```

namespace Parser_interface {
    double expr(bool);
}

```

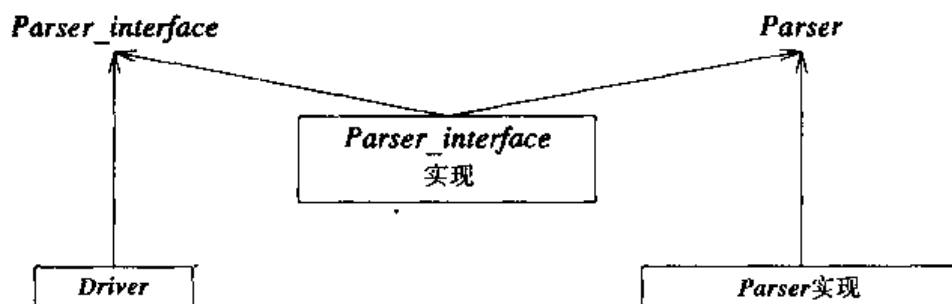
现在再去定义`Parser_interface`时, `Parser`已不必在作用域里了, 只有在定义`Parser_interface::expr()`的时候它才需要在作用域里:

```

double Parser_interface::expr(bool get)
{
    return Parser::expr(get);
}

```

最后的这一版本可以图示为:



现在所有的依赖性都已经最小化。每个东西都是具体的和适当命名的。当然, 对于我所面对的大部分问题而言, 这一解决方案也太过分了。

8.2.5 避免名字冲突

名字空间就是为了表示逻辑结构。最简单的这类结构就是分清楚由一个人写的代码与另一个人写的代码。这种简单划分也可能具有极大的实际重要性。

当我们只使用一个单独的名字空间时, 从一些相互独立的部分组合起一个程序, 就可能遇到一些不必要的困难。问题在于原本假定分离的部分中可能已经定义了同样的名字。在组合进同一个程序时, 这些名字就会出现冲突。考虑

```
// my.h:
char f(char);
int f(int);
class String { /* ... */ };

// your.h:
char f(char);
double f(double);
class String { /* ... */ };
```

有了这些定义之后，第三方很难同时使用`my.h`和`your.h`。一个明显的解决办法是将每组声明包裹在它自己的名字空间里：

```
namespace My {
    char f(char);
    int f(int);
    class String { /* ... */ };
}

namespace Your {
    char f(char);
    double f(double);
    class String { /* ... */ };
}
```

现在我们就可以通过显式的限定（8.2.1节）、使用声明（8.2.2节）或者使用指令（8.2.3节），去使用`My`和`Your`里的声明。

8.2.5.1 无名名字空间

有时，将一组声明包裹在一个名字空间里就是为了避免可能的名字冲突，这一做法经常也是很有价值的。这样做的目的只是保持代码的局部性，而不是为用户提供界面。例如，

```
#include "header.h"
namespace Mine {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

因为我们并不希望名字`Mine`被局部的环境之外知道，去发明这个多余的名字也就成了一种烦恼，还可能偶然地与其他什么名字相冲突。在这种情况下，我们可以简单地让这个名字空间没有名字：

```
#include "header.h"
namespace {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

很显然，必须存在某种方式，使我们可以从一个无名的名字空间之外访问其中的成员。因此，无名名字空间有一个隐含的使用指令[Ⓔ]。上面声明等价于

```
namespace $$$ {
    int a;
    void f() { /* ... */ }
    int g() { /* ... */ }
}
```

Ⓔ 这使无名名字空间里声明的所有东西在此之后可以直接使用（在本编译单位中）。如果没有这个规定，在这个无名的名字空间之后就再也无法使用`f()`和`g()`了，甚至无法再去定义它们。——译者注

```

}
using namespace $$$

```

其中\$\$\$是在这个名字空间定义所在的作用域里具有惟一性的名字。特别地，在不同编译单位里的无名名字空间也互不相同。正如我们所希望的，在其他编译单位里将无法说出这个无名名字空间中的成员名字。

8.2.6 名字查找

一个取*T*类型参数的函数常常与*T*类型本身定义在同一个名字空间里。因此，如果在使用一个函数的环境中无法找到它，我们就去查看它的参数所在的名字空间。例如，

```

namespace Chrono {
    class Date { /* ... */ };

    bool operator==(const Date&, const std::string&);
    std::string format(const Date&);    // 做出字符串表示
    // ...
}

void f(Chrono::Date d, int i)
{
    std::string s = format(d);          // Chrono::format()
    std::string t = format(i);          // 错误：在作用域里没有format()
}

```

与显式地使用限定相比，这个查找规则能使程序员节省许多输入，而且又不会以使用指令（8.2.3节）那样的方式污染名字空间。这个规则对于运算符的运算对象（11.2.4节）和模板参数（C.13.8.4节）特别有用，因为在那里使用显式限定是非常麻烦的。

请注意，名字空间本身必须在作用域里，而函数也必须在它被寻找和使用之前声明。

当然，一个函数的参数可以来自多个名字空间。例如，

```

void f(Chrono::Date d, std::string s)
{
    if (d == s) {
        // ...
    }
    else if (d == "August 4, 1914") {
        // ...
    }
}

```

对于这种情况，我们将在调用的作用域里（与往常一样），在每个参数（包括每个参数的类和基类）的名字空间里查找函数，而后对找到的所有函数执行普通的重载解析规则（7.4节）。特别地，对于上面的调用`d == s`，我们将在围绕`f()`的作用域里，在`std`名字空间里（这里有对于`string`的`==`定义），以及在`Chrono`名字空间里查找`operator==`。存在一个`std::operator==`，但它不以`Date`为参数；所以我们用`Chrono::operator==()`，它确实能用。另见11.2.4节。

当一个类的成员调用一个命名函数时，函数查找时应当偏向于同一个类及其基类的其他成员，而不是基于其他参数的类型可能发现的函数。对运算符的情况则有些不同（11.2.1节、11.2.4节）。

8.2.7 名字空间别名

如果用户给他们的名字空间取很短的名字，不同名字空间的名字也可能出现冲突：

```
namespace A { // 短名字, (最终) 将冲突
    // ...
}

A::String s1 = "Grieg";
A::String s2 = "Nielsen";
```

然而长名字在实际代码中又很不实用:

```
namespace American_Telephone_and_Telegraph { // 太长
    // ...
}

American_Telephone_and_Telegraph::String s3 = "Grieg";
American_Telephone_and_Telegraph::String s4 = "Nielsen";
```

这种两难境地可以通过为长名字提供较短的别名的方式解决:

```
// 为名字空间提供较短的别名:

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "Grieg";
ATT::String s4 = "Nielsen";
```

名字空间别名也使用户能够引用“某一个库”，并通过惟一的一个声明来定义那个库到底是什么。例如，

```
namespace Lib = Foundation_library_v2r11;

// ...

Lib::set s;
Lib::String s5 = "Sibelius";
```

这将使得用库的一个版本取代另一个的工作得到极大的简化。通过使用`Lib`而不是直接用`Foundation_library_v2r11`，当你需要更新到版本“v3r02”时，只要修改别名`Lib`的初始化并重新编译。当然，在另一方面，过多使用别名（无论什么形式）也会引起混乱。

8.2.8 名字空间组合

我们常常需要从现存的界面出发组合出新的界面，例如，

```
namespace His_string {
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
    void fill(char);
    // ...
}

namespace Her_vector {
    template<class T> class Vector { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_string;
    using namespace Her_vector;
    void my_fct(String&);
}
```

有了这些，我们就可以在**My_lib**的基础上写程序了：

```
void f()
{
    My_lib::String s = "Byron";    // 找到My_lib::His_string::String
    // ...
}

using namespace My_lib;

void g(Vector<String>& vs)
{
    // ...
    my_fct(vs[5]);
    // ...
}
```

如果显式限定的名字（例如，**My_lib::String**）在所说的名字空间里没有声明，编译器就会去查看使用指令说到的名字空间（例如，**His_string**）。

仅当我们需要去定义什么东西时，才需要知道一个实体真正所在的名字空间：

```
void My_lib::fill(char c)    // 错误：在My_lib里没有声明fill()
{
    // ...
}

void His_string::fill(char c)    // 可以：fill()在His_string里声明
{
    // ...
}

void My_lib::my_fct(String& v)    // 可以：String是My_lib::String，意思是His_string::String
{
    // ...
}
```

按理想情况，一个名字空间应该：

- [1] 描述了一个具有逻辑统一性的特征集合。
- [2] 不为用户提供对无关特征的访问。
- [3] 不给用户强加任何明显的记述负担。

这里和下面几小节中给出的组合技术——与 **#include** 机制（9.2.1节）一起——对上述思想提供了强有力的支持。

8.2.8.1 选择

有时我们只是想从一个名字空间里选用几个名字。我们可以做到这一点，只要写出一个仅仅包含我们想要的声明的名字空间。例如，我们可以声明一个**His_string**版本，它只提供了**String**本身和拼接运算符：

```
namespace His_string {           // 只有His_string的一部分
    class String { /* ... */ };
    String operator+(const String&, const String&);
    String operator+(const String&, const char*);
}
```

然而，除非我就是**His_string**的设计师或维护者，否则这一做法很快就会陷入混乱。任何对

“真正的” *His_string* 定义的修改都不会在这个声明中反映出来。通过使用声明来做，可以使从名字空间里选择一些特征的事变得更加明确：

```
namespace My_string {
    using His_string::String;
    using His_string::operator+; // 使用His_string的任何 +
}
```

使用声明将具有所指定的名字名的每个声明都带入作用域。特别地，通过一个使用声明就可以将一个重载函数的所有变形都带进来。

在这种方式下，如果 *His_string* 的维护者给 *String* 增加了一个成员函数，或者给拼接运算符增加了一个重载版本，这种修改将自动地变成 *My_string* 的用户可用的东西。相反地，如果某个特征被从 *His_string* 里删除，或者其界面修改了，对 *My_string* 使用的影响也将被编译器检查出来（另见 15.2.2 节）。

8.2.8.2 组合和选择

将组合（通过使用指令）和选择（通过使用声明）结合起来能产生更多的灵活性，这些也都是真实世界的例子所需要的。依靠这些机制，我们就能有这样的方式，它们既能提供对许多机制的访问，又能消解由于组合而产生的名字冲突或者歧义性。看例子，

```
namespace His_lib {
    class String { /* ... */ };
    template<class T> class Vector { /* ... */ };
    // ...
}

namespace Her_lib {
    template<class T> class Vector { /* ... */ };
    class String { /* ... */ };
    // ...
}

namespace My_lib {
    using namespace His_lib; // 来自His_lib的所有东西
    using namespace Her_lib; // 来自Her_lib的所有东西

    using His_lib::String; // 以偏向His_lib的方式解析潜在的冲突
    using Her_lib::Vector; // 以偏向Her_lib的方式解析潜在的冲突

    template<class T> class List { /* ... */ }; // 增加的东西
    // ...
}
```

在查看一个名字空间时，其中显式声明的名字（包括通过使用声明声明的名字）优先于在其他作用域里的那些通过使用指令才能访问的名字（C.10.1 节）。这样，*My_lib* 的用户将会看到，对于 *String* 和 *Vector* 名字冲突的解析将分别偏向于 *His_lib::String* 和 *Her_lib::Vector*。还有，*My_lib::List* 将总会被使用，与 *His_lib* 或 *Her_lib* 是否提供了 *List* 完全没关系。

通常，在将一个名字引进一个新的名字空间时，我最喜欢让它保持不变。按照这种方式，我就不必去记住两个不同的名字实际上指的是同一个东西。当然，有时确实也会需要新名字，或者用新的更好。例如，

```
namespace Lib2 {
    using namespace His_lib; // 来自His_lib的所有东西
```

```

using namespace Her_lib; // 来自Her_lib的所有东西

using His_lib::String;    // 以偏向His_lib的方式解析潜在的冲突
using Her_lib::Vector;    // 以偏向Her_lib的方式解析潜在的冲突

typedef Her_lib::String Her_string;    // 重命名

template<class T> class His_vec          // “重命名”
    : public His_lib::Vector<T> { /* ... */ };

template<class T> class List { /* ... */ }; // 增加的东西
// ...
}

```

这里不存在专用于重命名的特定语言机制，使用的是定义新实体的通用机制。

8.2.9 名字空间和老代码

数百万行的C和C++代码依赖于全局的名字和各种现存的库。我们如何利用名字空间来缓和这种代码中的问题呢？重新设计现存代码当然并不总是可行的选择。幸运的是，我们还是可能继续使用C库，就像它们是在名字空间里定义的一样。然而，对于C++写的库则无法做到这一点（9.2.4节）。在另一方面，在设计名字空间时也考虑了这方面的问题，设法尽可能地减小由于它的引进给已有C++程序带来的破坏。

8.2.9.1 名字空间和C

考虑常规的第一个C程序

```

#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
}

```

打破这个程序当然不是一个好主意。将标准库当做特例同样也不是好主意。因此，语言对名字空间的规则采用了这样一种设计，对一个没有用名字空间的程序，将它转换为使用名字空间的更清晰结构的过程相对来说比较容易完成。事实上，计算器程序（6.1节）也就是这样的一个例子。

使用指令是达到这一点的关键。例如，标准C中来自C头文件*stdio.h*的I/O功能被包裹在如下形式的名字空间里：

```

// stdio.h:

namespace std {
    int printf(const char* ... );
    // ...
}

using namespace std;

```

这样就得到了向后兼容性。还有，这里又定义了一个新头文件*cstdio*，提供给那些不希望一大批名字都能隐式地随便使用的人们：

```

// cstdio:

namespace std {
    int printf(const char* ... );
}

```



```

    // ...
}

```

当然，担心重复声明问题的C++ 标准库实现者可以通过包含*cstdio*的方式实现*stdio.h*：

```

// stdio.h:

#include <cstdio>
using std::printf;
// ...

```

我基本上把非局部的使用指令只看做是一种转变的工具。引用了来自其他名字空间中各种名字的代码，大都能够通过显式的限定和使用声明表述得更加清晰。

关于名字空间和连接之间的关系将在9.2.4节描述。

8.2.9.2 名字空间和重载

重载可以跨名字空间工作。对于我们能以最小的代价将现存的库修改为使用名字空间的东西而言，这特征是必不可少的。看下面例子：

```

// 老的A.h

void f(int);
// ...

// 老的B.h

void f(char);
// ...

// 老的user.c

#include "A.h"
#include "B.h"

void g()
{
    f('a'); 调用B.h里的f()
}

```

这个程序可以升级到使用名字空间的版本，无需修改实际代码：

```

// 新的A.h:

namespace A {
    void f(int);
    // ...
}

// 新的B.h:

namespace B {
    void f(char);
    // ...
}

// 新的user.c:

#include "A.h"
#include "B.h"

using namespace A;
using namespace B;

```

```
void g()
{
    f('a'); // 调用B.h里的f()
}
```

如果希望保持`user.c`完全不改变，我们也可以把使用指令放进头文件里。

8.2.9.3 名字空间是开放的

名字空间是开放的；也就是说，你可以通过多个名字空间声明给它加入名字。例如，

```
namespace A {
    int f(); // 现在A有成员f()
}

namespace A {
    int g(); // 现在A有成员f()和g()
}
```

通过这种方式，我们就能支持将一个名字空间中放入几个大的程序片段，其方式就像老的库和应用存在于同一个全局名字空间中那样。为做到这些，我们必须允许一个名字空间的定义分布到多个头文件和源代码文件里。正如前面计算器的例子（8.2.4节）所示，名字空间的开放性使我们可以通过展示名字空间不同部分的方式，为不同种类的用户提供不同的界面。这种开放性对于转变也很有帮助。例如，

```
// 我的头文件
void f(); // 我的函数
// ...
#include<stdio.h>
int g(); // 我的函数
// ...
```

可以重写，且不必重排声明的顺序：

```
// 我的头文件

namespace Mine {
    void f(); // 我的函数
    // ...
}

#include<stdio.h>

namespace Mine {
    int g(); // 我的函数
    // ...
}
```

在写新代码时，我倾向于使用较小的名字空间（8.2.8节），而不是将代码的主要片段都塞进同一个名字空间里。然而，在将一个软件的主要片段转为采用名字空间时，这种做法通常就不实际了。

在定义一个名字空间里已经声明过的成员时，更安全的方式是采用`Mine::`语法形式，而不是重新打开`Mine`。例如，

```
void Mine::ff() // 错误：在Mine里没有声明ff()
{
    // ...
}
```

编译器能够捕捉到这个错误。然而，由于在一个名字空间里可以定义新函数，编译器就无法捕捉在重新打开的名字空间里出现的同样错误

```
namespace Mine { // 重新打开Mine以定义函数
    void ff() // 呜呼！在Mine里没有声明ff()，这个定义将ff()加进了Mine
    {
        // ...
    }
    // ...
}
```

编译器没办法知道你并不想要这个新的`ff()`。

名字空间别名（8.2.7节）也可以在定义中用做名字的限定词。但是却不能通过名字空间的别名去重新打开那个名字空间。

8.3 异常

当一个程序是由一些相互分离的模块组成时，特别是当这些模块来自某些独立开发的库时，错误处理的工作就需要分成两个相互独立的部分：

[1] 一方报告出那些无法在局部解决的错误。

[2] 另一方处理那些在其他地方检查出的错误。

一个库的作者可以检查出运行时的错误，但一般说，他对于应该如何去做就没有什么主意了。库的使用者可能知道如何去处理某些错误，但却无法去检查它们——要不然用户就会在自己的代码里处理这些错误，而不会把它们留给库。

在计算器的例子里我们回避了这个问题，因为那里设计的是一个程序整体。在这样做时，我们就可能将错误处理问题放在整个程序的框架中考虑。但是，当我们将计算器程序的各个逻辑部分分开，放进各自的名字空间之后，我们就看到每个名字空间都要依赖于名字空间`Error`（8.2.2节），而`Error`里的错误处理又要依赖于各模块在错误出现后的恰当行为。现在假定我们并没有足够的自由，无法去整体地设计这个计算器，也不希望在`Error`和其他模块之间有过强的联系。让我们做一个相反的假定，假定我们在写分析器等的时候并不知道驱动程序可能怎样处理错误。

虽然`error()`非常简单，但它也包含了一种处理错误策略：

```
namespace Error {
    int no_of_errors;

    double error(const char* s)
    {
        std::cerr << "error: " << s << '\n';
        no_of_errors++;
        return 1;
    }
}
```

函数`error()`写出一条错误信息，提供一个默认值，使调用它的程序能继续计算下去，并保存一个简单的错误状态轨迹。更重要的是，程序中的每个部分都知道`error()`的存在，知道如何去调用它，以及可以期望由它得到些什么。对于一个由分别开发的库组合起来的程序而言，假定这么多东西或许是太多了。

异常机制是C++ 中用于将错误报告与错误处理分离开的手段。在这一节里，我们要在把异常应用于计算器实例的环境中给出有关异常的简单描述。第14章将提供有关异常及其使用的更广泛的讨论。

8.3.1 抛出和捕捉

提供异常这个概念就是为了帮助处理错误的报告。例如，

```
struct Range_error {
    int i;
    Range_error(int ii) { i = ii; } // 构造函数 (2.5.2节、10.2.3节)
};

char to_char(int i)
{
    if (i < numeric_limits<char>::min() || numeric_limits<char>::max() < i) // 参看22.2节
        throw Range_error(i);
    return i;
}
```

函数`to_char()`或者返回具有数值`i`的`char`，或者抛出一个`Range_error`。这里的基本想法是，如果函数发现了一个自己无法处理的问题，它就抛出（`throw`）一个异常，希望它的（直接或间接）调用者能够处理这个问题。如果一个函数想处理某个问题，它就可以说明自己要捕捉（`catch`）用于报告该种问题的异常。例如，如果想调用`to_char()`并捕捉它可能抛出的异常，我们可以写

```
void g(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
    catch (Range_error) {
        cerr << "oops\n";
    }
}
```

程序结构

```
catch ( /* ... */ ) {
    // ...
}
```

称为一个异常处理器，它的使用只能紧接着由`try`关键字作为前缀的块之后，或者紧接着另一个异常处理器之后。`catch`也是一个关键字。在括号里包含着一个声明，其使用方式类似于函数参数的声明。也就是说，该声明描述的是这个处理器要捕捉的对象的类型，并可以为所捕捉的对象命名。举个例子，如果我们想知道抛出的`Range_error`的值，我们就可以为`catch`的参数提供一个名字，就像为函数的参数命名一样。例如写：

```
void h(int i)
{
    try {
        char c = to_char(i);
        // ...
    }
```

```

    catch (Range_error x) {
        cerr << "oops: to_char(" << x.i << ")\\n";
    }
}

```

如果在一个`try`块里的任何代码——或者由那里调用的东西——抛出了一个异常，那就会逐个检查这个`try`块后面的处理器。如果所抛出的异常属于某个处理器所描述的类型，这个处理器就会被执行。如果`try`块未抛出异常，这些异常处理器都将被忽略，使该`try`块的活动方式就像是普通的块。如果抛出了一个异常，而又没有`try`块捕捉它，整个程序就将终止（14.7节）。

简而言之，C++ 异常处理是一种从一个函数里将控制传递到有意确定的一些代码的方式。如果需要，也可以同时把关于错误的某些信息传递给调用者。C程序员可以将异常处理想像为一种用于取代`setjmp/longjmp`（16.1.2节）的、具有良好行为方式的机制。在异常处理与类之间重要的相互作用将在第14章讨论。

8.3.2 异常的辨识

典型情况下，在一个程序里可能存在多种不同的运行时错误，我们可以将这些错误映射到一些具有不同名字的异常。我喜欢定义一些除了服务于异常处理之外没有其他用途的类，这使人不容易将它们的用途弄错。特别地，我绝不去用内部类型（例如，`int`）作为异常。因为，在一个大程序里，我没有有效的方法去确定`int`异常的无关使用，这样，我也就无法保证那些另有所图的使用不会与我的使用相互干扰。

我们的计算器（6.1节）必须处理两类运行时的错误：语法错误和企图除以0的错误。检查到企图除以0错误的代码不需要给处理器传递任何信息，因此，除以0的问题可以用一个简单的空类型表示：

```
struct Zero_divide {};
```

而在另一方面，处理器非常希望能得到一个有关出现了什么语法错误的指示。在这里我们就传递一个字符串

```

struct Syntax_error {
    const char* p;
    Syntax_error(const char* q) { p = q; }
};

```

为了记述上的方便，我给这个`struct`加进了一个构造函数（2.5.2节、10.2.3节）。

分析器的用户可以通过在`try`块之后附加两个处理器的方式，完成对这两种异常的辨识，在需要时控制就会进入适当的处理器。如果我们从一个处理器的“末端掉出去”，执行就将从整个的处理器列表之后继续下去：

```

try {
    // ...
    expr(false);
    // 当且仅当expr()没有导致任何异常，我们将到达这里
    // ...
}
catch (Syntax_error) {
    // 处理语法错误
}

```

```

catch (Zero_divide) {
    // 处理用零除的错误
}
// 如果expr()没有发生任何异常,或者是出现Syntax_error
// 或Zero_divide异常并被捕捉到(而且其处理器不return,
// 不抛出异常,也不以其他方式改变控制流),我们就能到达这里

```

处理器的表看起来就像一个开关语句,但是这里不需要**break**。处理器列表在语法与**case**列表不同,部分的原因也就在于此,另一个原因是指明每个处理器都是一个作用域(4.9.4节)。

一个函数不必捕捉所有可能的异常。例如,前面的try块就没有打算去捕捉可能由分析器的输入操作产生的异常。这些异常将简单地“穿过”这里,继续去查找某个带有合适处理器的调用者。

从语言的观点看,被考虑那个异常正好在其处理器的入口处进行处理,所以,在执行处理器期间所抛出的异常就必须由这个try块的调用者去处理。举个例子,下面的代码不会导致无穷循环:

```

class Input_overflow { /* ... */ };
void f()
{
    try {
        // ...
    }
    catch (Input_overflow) {
        // ...
        throw Input_overflow();
    }
}

```

异常处理器也可以嵌套。例如,

```

class XXII { /* ... */ };
void f()
{
    // ...
    try {
        // ...
    }
    catch (XXII) {
        try {
            // 某些复杂的东西
        }
        catch (XXII) {
            // 复杂处理器代码失败
        }
    }
    // ...
}

```

当然,在人们写出的代码中很少会看到这种嵌套,这更多的是表明了某种糟糕风格。

8.3.3 在计算器中的异常

有了基本的异常处理机制,我们现在可以重做6.1节计算器的例子,将运行中所发现的错

误的处理从计算器的基本逻辑中分离出来。这将导致一种程序组织结构，它更真实地反映了那些由互相分离的联系松散的部分构造出的程序的情况。

首先，可以删除`error()`。与此对应，分析器函数只知道发出错误信号所使用的类型：

```
namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}
```

分析器检查三种语法错误

```
Lexer::Token_value Lexer::get_token()
{
    using namespace std;    // 使用输入, isalpha() 等 (6.1.7节)
    // ...

    default:                // NAME, NAME =, 或者错误
        if (isalpha(ch)) {
            string_value = ch;
            while (input->get(ch) && isalnum(ch)) string_value.push_back(ch);
            input->putback(ch);
            return curr_tok=NAME;
        }
        throw Error::Syntax_error("bad token");
    }
}

double Parser::prim(bool get)    // 处理初等项
{
    // ...

    case Lexer::LP:
    {
        double e = expr(true);
        if (curr_tok != Lexer::RP) throw Error::Syntax_error("` ` expected");
        get_token();    // 吃掉'
        return e;
    }
    case Lexer::END:
        return 1;
    default:
        throw Error::Syntax_error("primary expected");
    }
}
```

在检查到一个语法错误时，代码就通过`throw`将控制传递到在某个（直接或间接的）调用者里定义的异常处理器去。`throw`运算符同时给处理器送去一个值。例如，

```
throw Syntax_error("primary expected");
```

将向处理器传递一个`Syntax_error`对象，其中包含着一个指向字符串 `"primary expected"` 的指针。这正是那个处理器所期望的。

报告除零错误不需要传递任何数据：

```
double Parser::term(bool get)    // 乘和除
{
    // ...
    case Lexer::DIV:
        if (double d = prim(true)) {
            left /= d;
            break;
        }
        throw Error::Zero_divide();
    // ...
}
```

现在定义驱动程序，使之能处理`Zero_divide`和`Syntax_error`异常。比如写：

```
int main(int argc, char* argv[])
{
    // ...
    while (*input) {
        try {
            Lexer::get_token();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) continue;
            cout << Parser::expr(false) << '\n';
        }
        catch (Error::Zero_divide) {
            cerr << "attempt to divide by zero\n";
            if (Lexer::curr_tok != Lexer::PRINT) skip();
            ++Driver::no_of_error;
        }
        catch (Error::Syntax_error e) {
            cerr << "syntax error: " << e.p << "\n";
            if (Lexer::curr_tok != Lexer::PRINT) skip();
            ++Driver::no_of_error;
        }
    }

    if (input != &cin) delete input;
    return no_of_errors;
}
```

除非是在表达式的最后，由表示结束的`PRINT`单词（即，分号或者换行符）引起的错误，否则`main()`将调用恢复函数`skip()`。函数`skip()`尝试着将分析器带入一个定义良好的状态，它采用的方式就是丢掉一系列字符，直到遇到一个换行符或者分号为止。函数`skip()`、`no_of_errors`和`input`是名字空间`Driver`的当然候选成员：

```
namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip();
}

void Driver::skip()
{
    no_of_errors++;
    while (*input) {    // 丢掉一些字符，直至遇到换行符或者分号
        char ch;
        input->get(ch);

        switch (ch) {
```



```

        case '\n':
        case ';':
            return;
    }
}
}

```

`skip()` 的代码的抽象层次比分析器代码更低, 这样做是有意的, 是为了避免它在处理由分析器报告的异常期间被来自分析器的异常套住。

我保留了统计出错次数并将这个数通过程序返回值报告的想法。知道一个程序是否遇到错误, 它是否能够从错误中恢复等, 这些都非常有价值。

我没有将 `main` 放进 `Driver` 名字空间。全局的 `main` 是整个程序的初启函数 (3.2 节), 出现在某个名字空间里的 `main()` 就没有特殊意义了。在实际规模的程序中, 应该将 `main()` 中的大部分代码移到 `Driver` 里的另一个独立函数里。

8.3.3.1 其他错误处理策略

原来的错误处理代码比采用异常之后的代码更短, 也更优美。但那却是通过程序各部分之间的紧密耦合而得到的。那种方式不能很好地用到由分别开发的库组合而成的程序中。

我们也可以考虑去掉专门的错误处理函数 `skip()`, 方式是在 `main()` 里引进一个状态变量。例如,

```

int main(int argc, char* argv[]) // 糟糕风格的例子
{
    // ...

    bool in_error = false;
    while (*Driver::input) {
        try {
            Lexer::get_token();
            if (Lexer::curr_tok == Lexer::END) break;
            if (Lexer::curr_tok == Lexer::PRINT) {
                in_error = false;
                continue;
            }
            if (in_error == false) cout << Parser::expr(false) << '\n';
        }
        catch (Error::Zero_divide) {
            cerr << "attempt to divide by zero\n";
            in_error = true;
        }
        catch (Error::Syntax_error e) {
            cerr << "syntax error: " << e.p << "\n";
            in_error = true;
        }
    }

    if (Driver::input != &std::cin) delete Driver::input;
    return Driver::no_of_errors;
}

```

我认为这是一个很坏的办法, 因为:

- [1] 状态变量是混乱和错误的一个常见根源, 特别是允许它们大量出现并影响程序中较大的片段时。特别地, 我认为使用 `in_error` 的 `main()` 版本不如使用 `skip()` 的版本好读。

[2] 一般说,使处理错误的代码与“正常”代码分离是一种很好的策略。

[3] 在导致错误的代码的同一个抽象层次上处理错误是非常危险的。完成错误处理的代码有可能又产生了引起错误处理的那个错误。我把下面问题留做练习:请说明在使用 `in_error` 的 `main()` 版本中怎么会出现这种情况(8.5[7])。

[4] 修改“正常”的代码,加上错误处理代码,所需的工作量比增加单独的错误处理例行程序更多。

异常处理的意图是去处理非局部的问题。如果一个错误可以在局部处理,那么(几乎)总应该这样做。例如,完全没有必要对“过多参数错误”使用异常机制:

```
int main(int argc, char* argv[])
{
    using namespace std;
    using namespace Driver;

    switch (argc) {
        case 1:                // 从标准输入读
            input = &cin;
            break;
        case 2:                // 从参数字符串读
            input = new istringstream(argv[1]);
            break;
        default:
            cerr << "too many arguments\n";
            return 1;
    }
    // 如前
}
```

第14章将进一步讨论有关异常的问题。

8.4 忠告

[1] 用名字空间表示逻辑结构; 8.2节。

[2] 将每个非局部的名字放入某个名字空间里,除了 `main()` 之外; 8.2节。

[3] 名字空间的设计应该让你能很方便地使用它,而又不会意外地访问了其他的无关名字空间; 8.2.4节。

[4] 避免对名字空间使用很短的名字; 8.2.7节。

[5] 如果需要,通过名字空间别名去缓和长名字空间名的影响; 8.2.7节。

[6] 避免给你的名字空间的用户添加太大的记法负担; 8.2.2节、8.2.3节。

[7] 在定义名字空间的成员时使用 `namespace::member` 的形式; 8.2.8节。

[8] 只在转换时,或者在局部作用域里,才用 `using namespace`; 8.2.9节。

[9] 利用异常去松弛“错误”处理代码和正常处理代码之间的联系; 8.3.3节。

[10] 采用用户定义类型作为异常,不用内部类型; 8.3.2节。

[11] 当局部控制结构足以应付问题时,不要用异常; 8.3.3.1节。

8.5 练习

1. (*2.5) 参照2.4节的 `Stack` 模块的风格,写一个 `string` 的双向链表模块。通过建立一个程序设

计语言名字的表来演习这个模块。为这个表提供一个`sort()`函数，并提供一个函数去反转表中字符串的顺序。

2. (*2) 取若干不太大的程序，它们用到至少一个没有用名字空间定义的库。修改它，使它对有关的库使用一个名字空间。提示：8.2.9节。
3. (*2) 利用名字空间将桌面计算器程序修改为2.4节的风格。不要使用全局的使用指令。记录下你在修改中所犯的错误。提出一些方法，以便在将来避免这些错误。
4. (*1) 写一个程序，其中一个函数里抛出一个异常，另一个捕捉它。
5. (*2) 写一个程序，它由一些互相调用直到第10层的函数组成。给每个函数一个参数，确定是在哪个层次中抛出了异常。让`main()`捕捉这些异常，并打印出捕捉到的那个异常。请不要忘记这种情况：一个异常也可以在抛出它的函数里被捕捉到。
6. (*2) 修改8.5[5]的程序，做一些实测：让一个堆栈类在某个地方抛出异常，看看捕捉异常的代价是否因为抛出位置在函数调用栈中的位置不同而不同。在每个函数中增加一个字符串对象，再重新做实测。
7. (*1) 找出8.3.3.1节的第一个`main()`版本中的错误。
8. (*2) 写一个函数，它或者是返回一个值，或者基于某个参数而抛出这个值。实测这两种方式在运行时间上的差异。
9. (*2) 利用异常修改8.5[3]的计算器版本。记录下你所犯的错误。提出一些方法，以便在将来避免这些错误。
10. (*2.5) 写出能够检查可能的上溢和下溢的`plus()`、`minus()`、`multiply()`、`divide()`函数，让它们在出现这些错误时抛出异常。
11. (*2) 修改计算器，使用8.5[10]定义的那些函数。

第9章 源文件和程序

形式必须与功能相符。

——Le Corbusier

分别编译——连接——头文件——标准库头文件——惟一定义规则——与非C++ 代码的连接——连接和指向函数的指针——利用头文件表达模块性——单一文件组织——多头文件组织——包含保护符——程序——忠告——练习

9.1 分别编译

文件是传统的（在文件系统里的）存储单位和传统的编译单位。也存在着一些不同的系统，它们并不按照程序员可以看到的一集文件的方式存储、编译和展现C++程序。当然，这里的讨论将集中关注更传统的使用文件的系统。

将一个完整的程序放入一个文件里通常是不可能的。特别是，在典型情况下，标准库和操作系统的代码都不是以源程序形式提供的，不能作为用户程序的一部分。对于具有真实规模的程序，将用户自己的所有代码放入一个文件也很不实际、不方便。采取将程序组织为一些文件的方式，可以强调它的逻辑结构，帮助读者理解程序，并帮助编译器去强迫实施这种逻辑结构。如果编译的单位就是一个文件，那么，当该文件或者它所依赖的什么东西在某个时候做了修改（哪怕非常小），这个文件的整体就必须重新编译。即使对一个中等规模的程序，将它划分为一些适当大小的文件，所节约的重编译时间也可能非常可观。

用户将一个源文件提交给编译器后，首先进行的是该文件的预处理，也就是说，完成宏处理（7.8节），并按照 `#include` 指令引进所有头文件（2.4.1节、9.2.1节）。预处理之后的结果被称为编译单位。这种编译单位才是编译器真正的工作对象，也是C++语言的规则所描述的对象。在本书中，如果需要区分程序员所看到的東西和编译器所看到的東西时，我就会分别说源文件和编译单位。

为了使分别编译能够工作，程序员必须提供各种声明，为孤立地分析一个编译单位提供有关程序其他部分的类型信息。在一个由许多分别编译的部分组成的程序里，这些声明必须保持一致，就像在由一个编译单位组成的程序里，所有声明都必须一致一样。你的系统中可能有某些工具能帮助你保证这一点。特别是连接器能够检查出许多种类的不一致性。连接器是一个程序，它的工作就是将分别编译的部分约束在一起。连接器有时也被（有些混乱地）称做装载器（loader）。连接工作可以在程序开始运行之前完全做好。另一种方式是允许在程序开始运行后为其加入新代码（动态连接）。

程序是由一些文件组成的，这种组织通常被称为程序的物理结构。将一个程序物理地划分为一些相互分离的文件的工作应该根据程序的逻辑结构进行。对依赖性问题的关注指导着我们由一些名字空间组合出程序，同样的关注也指导着由源文件构成程序的组合过程。当然，

一个程序的逻辑结构和物理结构不必完全相同。例如，可以用若干个源程序文件存储某一个名字空间里的函数，或者将几个名字空间定义放入同一个文件里，或者把一个名字空间的定义散布在若干个文件里，这些方式都很有价值（8.2.4节）。

在这里，我们将首先考虑一些有关连接的技术细节，而后再讨论将计算器程序（6.1节、8.2节）分割成多个文件的两种方式。

9.2 连接

在所有的编译单位中，对所有函数、类、模板、变量、名字空间、枚举和枚举符的名字的使用都必须保持一致，除非它们被显式地描述为局部的东西。

所有名字空间、类、函数等都应该在它们出现的各个编译单位中有适当的声明，而且所有声明都应该一致地引用同一个实体，保证这一点是程序员的工作。例如，考虑两个文件

```
// file1.c:
    int x = 1;
    int f() { /* 做某些事情 */ }

// file2.c:
    extern int x;
    int f();
    void g() { x = f(); }
```

在`file2.c`里使用的`x`和`f()`就是在`file1.c`里定义的那些东西。关键字`extern`指明在`file2.c`里`x`的声明（只）是一个声明，而不是一个定义（4.9节）。如果在这里出现`x`的初始式，那么就简单地忽略`extern`，因为带有初始式的声明就是定义。在一个程序里，一个对象只能定义一次，它可以有多个声明，但类型必须完全一样。例如，

```
// file1.c:
    int x = 1;
    int b = 1;
    extern int c;

// file2.c:
    int x;           // 意思是int x = 0;
    extern double b;
    extern int c;
```

这里存在着3个错误：`x`定义了两次，`b`用不同的类型定义了两次，而`c`只声明了两次但没有定义。一个每次只看一个文件的编译器将无法检查出这些种类的错误（连接错误），这些错误中的大部分可以被连接器检查出来。请注意，如果定义在全局作用域或者名字空间作用域里某一个变量没有初始式，它就会被按照默认方式初始化。对局部变量（4.9.5节、10.4.2节）和在自由存储中建立的对象（6.2.6节）而言，情况就不是这样。例如，

```
// file1.c:
    int x;
    int f() { return x; }

// file2.c:
    int x;
    int g() { return f(); }
```

程序片段中包含两个错误。

在`file.c`里调用`f()`是个错误,因为`f()`没有在`file2.c`里声明。同样,程序也无法连接,因为`x`定义了两次。注意,在C语言中这个`f()`调用不是错误(B.2.2节)。

如果一个名字可以在与其定义所在的编译单位不同的地方使用,就说它是具有外部连接的。前面例子里的所有名字都具有外部连接。如果某个名字只能在其定义所在的编译单位内部使用,它就被称为是具有内部连接的。

一个`inline`函数(7.1.1节、10.2.9节)必须在需要用它的每个编译单位里定义——通过完全一样的定义(9.2.3节)。因此,下面的例子不过是一个极坏的尝试,它是非法的:

```
// file1.c:
inline int f(int i) { return i; }

// file2.c:
inline int f(int i) { return i+1; }
```

不幸的是,这种错误却很难被具体实现检查出来。也正是因为这种情况,另一种本来具有完美逻辑的东西——外部连接和在线的组合——只好也禁止了,以便使写编译器的人们的工作更简单一些:

```
// file1.c:
extern inline int g(int i);
int h(int i) { return g(i); } // 错误: g() 在这个编译单位里无定义

// file2.c:
extern inline int g(int i) { return i+1; }
```

按照默认约定,`const`(5.4节)和`typedef`(4.9.7节)都具有内部连接。因此下面的例子就是合法的(虽然很可能把人弄糊涂):

```
// file1.c:
typedef int T;
const int x = 7;

// file2.c:
typedef void T;
const int x = 8;
```

局部于一个编译单位的全局变量是造成混乱的一个常见根源,最好是避免之。为了保证一致性,你一般应该把全局的`const`和`inline`都仅仅放在头文件里(9.2.1节)。

通过显式声明可以使`const`具有外部连接:

```
// file1.c:
extern const int a = 77;

// file2.c:
extern const int a;

void g()
{
    cout << a << '\n';
}
```

在这里`g()`将打印出77。

无名名字空间(8.2.5节)可以用于使一些名字局部于一个编译单位。无名名字空间的效果很像是内部连接。例如,

```
// file1.c:
namespace {
    class X { /* ... */ };
    void f();
    int i;
    // ...
}

// file2.c:
class X { /* ... */ };
void f();
int i;
// ...
```

位于`file1.c`里的函数`f()`与`file2.c`里的`f()`不是同一个函数。让一个名字局部于某个编译单位，而又将同一个名字用在其他地方，表示一个具有外部连接的实体，这样做实在是自找麻烦。

在C和C++程序里，关键字`static`也被（有些混乱地）用于表示“使用内部连接”（B.2.3节）。请不要使用`static`，除了在函数（7.1.2节）和类（10.2.4节）的内部。

9.2.1 头文件

对同一个对象、函数、类等的所有声明的类型都必须一致。所以，递交给编译器的源代码以及后来被连接的东西也必须一致。要达到在不同编译单位中声明的一致性，有一种不完美但却比较简单的方法，那就是将包含界面信息的头文件通过 `#include` 包含到可执行代码和/或数据定义的源程序文件里。

`#include` 机制是一种正文操作的概念，用于将源程序片段收集到一起，形成一个提供给编译的单位（文件）。指令

```
#include "to_be_included"
```

将用文件`to_be_included`的内容取代这个 `#include` 所在的那一行。该文件的内容应该是C++ 源代码，因为编译器接着要去读它。

如果要包含标准库头文件，那么就应使用尖括号 `<` 和 `>` 而不是引号。例如，

```
#include <iostream>    // 来自标准库包含目录
#include "myheader.h"  // 来自当前目录
```

不幸的是，在一个包含指令中，`<>` 或 `" "` 内部的空格也是有意义的：

```
#include < iostream >    // 将无法找到 <iostream>
```

当一个文件被包含到某处之后，每次都需要重新去编译它，这看起来是过分奢侈了。不过，在典型情况下被包含的文件里只有声明，没有需要编译器深入分析的代码。进一步说，许多现代C++ 实现都提供了对头文件的某种预编译形式，以尽可能减少反复编译同一个头文件所需要的工作。

作为一种经验法则，头文件里可以包括：

命名名字空间	<code>namespace N { /* ... */ }</code>
类型定义	<code>struct Point { int x, y; };</code>
模板声明	<code>template <class T> class Z;</code>
模板定义	<code>template <class T> class V { /* ... */ };</code>

(续)

函数声明	<code>extern int strlen (const char*);</code>
在线函数定义	<code>inline char get (char* p) {return *p++; }</code>
数据声明	<code>extern int a;</code>
常量定义	<code>const float pi = 3.141593;</code>
枚举	<code>enum Light { red, yellow, green };</code>
名字声明	<code>class Matrix;</code>
包含指令	<code>#include <algorithm></code>
宏定义	<code>#define VERSION 12</code>
条件编译指令	<code>#ifdef __cplusplus</code>
注释	<code>/* check for end of file */</code>

这种有关什么可以放入头文件的经验法则并不是语言所要求的。它只是使用 `#include` 机制来表示程序的物理结构的一种合理方式。在另一方面，头文件里绝不应该有：

常规的函数定义	<code>char get (char* p) { return *p++; }</code>
数据定义	<code>int a;</code>
聚集量 (aggregate) 定义	<code>short tbl[] = { 1, 2, 3 };</code>
无名名字空间	<code>namespace { /* ... */ }</code>
导出的 (exported) 模板定义	<code>export template <class T> f (T t) { /* ... */ }</code>

按照习惯，头文件采用后缀 `.h`，而包含函数或数据定义的文件用 `.c` 后缀。它们也因此被分别称为“`.h` 文件”和“`.c` 文件”。其他约定，如 `.C`、`.cxx`、`.cpp` 和 `.cc` 等也常常可以看到。你的编译器手册在这个方面可能有一些特殊东西。

上面建议只在头文件放简单常量的定义，而不放聚集量的定义，其原因是，具体实现很难避免聚集量在几个编译单位中的重复出现。进一步说，那些简单情况都是最常见的东西，因此对于生成好的代码也更重要一些。

对于 `#include` 的使用不要过于自作聪明。我的建议是，只用 `#include` 包含完整的声明和定义，而且只在全局作用域中，或在连接描述块里，或在转换老代码时在名字空间定义里这样做（9.2.2 节）。与别处一样，在这里也要避免玩弄宏魔术。我最不愿意做的事情之一就是：追踪由某个名字引起的一个错误，该名字被宏替换成另一个完全不同的东西，而有关的宏定义又是出现在某个我从来也没听说过的，通过 `#include` 间接包含进来的头文件里。

9.2.2 标准库头文件

标准库的功能是通过一组标准头文件给出的（16.1.2 节）。标准库头文件不需要后缀；它们被当做头文件是因为它们需要用 `#include<...>` 包含进来，而不是用 `#include"..."`^①。缺少 `.h` 后缀并不意味着这些头文件必须采用某种特殊的存储方式。举例来说，头文件 `<map>` 有可能作为一个名字是 `map.h` 的正文文件存储在某个标准目录里。在另一方面，标准头文件也不必按常规方式存储。一个具体实现可以利用有关标准库定义的知识，去优化标准库的实现以及对标准头文件的处理方式。比如说，某个实现可以将标准数学库（22.3 节）做成内部的，且将 `#include<cmath>` 当做一个开关，在使数学库能够使用的同时又不需要读入任何文件。

① 原文如此。这个解释似乎不能说明问题。头文件是一种概念，被用作共享性信息的承载物，以便在不同的物理的程序文件之间维护某种一致性关系。标准库头文件就是标准库提供的一组头文件，语言的标准为它们确定了目前的这一套命名方式，参见第16章。——译者注

相对于每一个C标准库文件 `<X.h>`，存在着一个与之对应的标准C++ 头文件 `<cX>`。例如，`#include <cstdio>` 提供的是与 `#include <stdio.h>` 同样的东西。一个典型的 `<stdio.h>` 看起来是某种类似下面的东西

```
#ifndef __cplusplus           // 只为了C++编译器 (9.2.4节)
namespace std {              // 标准库定义在名字空间std里 (8.2.9节)
extern "C" {                  // 在stdio里的函数具有C连接 (9.2.4节)
    #endif

    /* ... */
    int printf(const char* ...);
    /* ... */

#ifdef __cplusplus
}
}
using namespace std;         // 使Printf在全局名字空间里可以使用
#endif
```

也就是说，实际声明（基本上）都是共享的，但连接和名字空间问题则需要另行处理，以便使C和C++ 程序能共享这个头文件。

9.2.3 单一定义规则

在一个程序里，任一个类、枚举和模板等必须只定义惟一的一次。

从实践的观点看，这意味着对（比如说）一个类，必须在某个单独的文件里存在着恰好一个类定义。不幸的是，语言的规则不能这么简单。例如，某个类的定义可能是通过宏展开（呜呼！）组合而成的；某个类的定义也可能通过 `#include` 指令，以文本方式被包含到两个源文件里（9.2.1节）。更糟糕的是，“文件”并不是C或C++ 语言定义的一部分；存在着一些实现，它们根本不将程序存储为源文件。

因此，标准中关于一个类、一个模板等只能有惟一定义的规则，就需要以一种更复杂更精细的方式来陈述。这个规则经常被称为“单一定义规则”（One-Definition Rule, ODR）。也就是说：一个类、模板或者在线函数的两个定义能够被接受为同一个惟一定义的实例，当且仅当：

- [1] 它们出现在不同编译单位里。
- [2] 它们按一个个单词对应相同。
- [3] 这些单词的意义在两个编译单位中也完全一样。

例如，

```
// file1.c:
struct S { int a; char b; };
void f(S*);

// file2.c:
struct S { int a; char b; };
void f(S* p) { /* ... */ }
```

ODR认为这个例子是合法的，而且S在两个源文件里引用了同一个类。然而，像这样两次写出一个定义很不聪明。维护file2.c的人们将会很自然地假定在file2.c里的S的定义就是S的惟一定义，因此认为可以自由地去修改它。这样就会引进难以检查的错误。

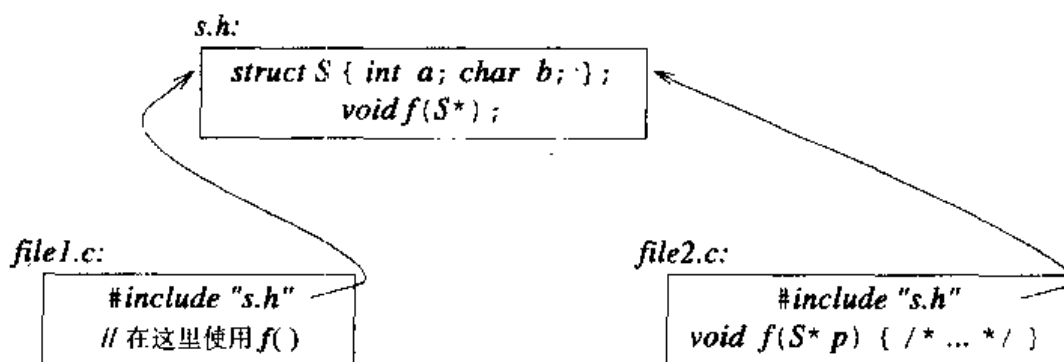
ODR的意图是想允许将一个公共源文件里的类定义包含到不同的编译单位里去。例如，

```
// file s.h:
    struct S { int a; char b; };
    void f(S*);

// file1.c:
    #include "s.h"
    // 在这里使用 f()

// file2.c:
    #include "s.h"
    void f(S* p) { /* ... */ }
```

或图示为



下面给出违反ODR的三种情况的例子：

```
// file1.c:
    struct S1 { int a; char b; };
    struct S1 { int a; char b; }; // 错误：重复定义
```

这是错误的，因为在一个编译单位里，同一个`struct`不能定义两次。

```
// file1.c:
    struct S2 { int a; char b; };

// file2.c:
    struct S2 { int a; char bb; }; // 错误
```

这是错误的，因为S2被用做两个成员名不相同的类的名字。

```
// file1.c:
    typedef int X;
    struct S3 { X a; char b; };

// file2.c:
    typedef char X;
    struct S3 { X a; char b; }; // 错误
```

这里的两个S3定义按单词对应相同。但这个例子是错误的，因为在两个文件里，名字X的意义被偷偷摸摸地做成不同的了。

在互相分离的编译单位里检查和抵御不一致的类定义，这件事情超出了许多实现的能力范围。因此，违背ODR的声明就可能成为难以琢磨的错误的根源。不幸的是，将共享定义放在头文件并 `#include` 它们的技术并不能防范上面的最后一种违背ODR的形式。局部`typedef`和宏都可能改变通过 `#include` 包含进来的声明的意义：

```
// file s.h:
    struct S { Point a; char b; };

// file1.c:
    #define Point int
    #include "s.h"
    // ...

// file2.c:
    class Point { /* ... */ };
    #include "s.h"
    // ...
```

抵御这类黑客手段的最好办法就是将头文件尽可能做得自给自足。例如，如果类**Point**已经在**s.h**头文件里声明了，上面这个错误就会被检查出来。

也允许将一个模板定义 **#include** 到多个编译单位里，条件是仍能维持ODR。另外，如果只存在一个声明，就可以使用一个导出的 (**export**) 模板

```
// file1.c:
    export template<class T> T twice(T t) { return t+t; }

// file2.c:
    template<class T> T twice(T t);           // 声明
    int g(int i) { return twice(i); }
```

关键字**export**的意思就是“在其他的编译单位可以访问”。

9.2.4 与非C++代码的连接

在许多情况下，一个C++ 程序中也可能包含着一些采用其他语言写出的部分。类似地，C++ 代码片段也经常用于主要由其他语言写出的程序。用不同语言写出的程序片段之间的协作比较困难；甚至采用同一种语言写出，但通过不同编译器编译的片段也是这样。例如，不同语言，或者同一语言的不同实现可能在它们使用寄存器保存参数的方式上，在将参数放入堆栈的顺序上，在整数或字符串等内部类型的布局上，在编译器传递给连接器的名字方面，在对连接器所要求的类型检查的量等方面存在着差异。为了能有所帮助，可以在一个**extern**声明中给出有关的连接约定。例如，下面声明了C和C++ 标准库函数**strcpy()**，并特别说明它应该按照C连接约定进行连接：

```
extern "C" char* strcpy(char*, const char*);
```

这个声明的作用与

```
extern char* strcpy(char*, const char*);
```

“简单”声明的差异仅在于调用**strcpy()**的连接约定不同。

由于C和C++ 语言之间的紧密关系，**extern "C"** 指令就特别有用。请注意，在**extern "C"** 中的**C**表示一个连接约定，而不是一种语言。人们也经常将**extern "C"** 用于连接Fortran和汇编例程，因为它们的要求也正好符合C实现的约定。

extern "C" 指令描述的 (只) 是一种连接约定，它并不影响调用函数的语义。特别地，声明为**extern "C"** 的函数仍然要遵守C++ 的类型检查和参数转换规则，而不是C的较弱的规则。例如，

```
extern "C" int f();

int g()
```

```
{
    return f(1);    // 错误：未期望有参数
}
```

为许多声明添加**extern "C"** 也会成为令人讨厌的事情。因此，这里还提供了一种为一组声明描述连接约定的机制。例如，

```
extern "C" {
    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...
}
```

这种结构经常被称为连接块，它可以用于包裹起整个的C头文件，使整个文件能适合C++的使用。例如，

```
extern "C" {
#include <string.h>
}
```

这种技术经常被用于由C头文件产生出的C++ 头文件。换一种方式，也可以用条件编译（7.8.1节）建立起公共的C和C++ 头文件：

```
#ifdef __cplusplus
extern "C" {
#endif

    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...

#ifdef __cplusplus
}
#endif
```

_cplusplus是一个预定义的宏名字，它被用于保证当这个文件被用做C头文件时，其中的C++结构将被去掉。

任何声明都可以出现在连接块里：

```
extern "C" {           // 这里可以有任意声明，如：
    int g1;           // 定义
    extern int g2;    // 声明，不是定义
}
```

应特别指出，变量的作用域及存储类都不会受到影响。所以，**g1**仍然是一个全局变量，还是在这里被定义而不是被声明。要想声明而不是定义一个变量，你必须将关键字**extern**直接应用在声明里。例如，

```
extern "C" int g3;      // 声明，不是定义
```

这一写法初看起来有点古怪。然而，它不过是给一个外部声明加上**"C"**后应该保持意义不变，将一个文件用连接块包裹后意义也不变的一个简单推论。

有着C连接的名字也可以在名字空间里定义。名字空间只影响到C++ 程序里对于该名字的访问方式，但却不会影响连接器看到它的方式。取自**std**的**printf()**是一个典型的例子：

```
#include<stdio>

void f()
{
    std::printf("Hello, "); // ok
    printf("world!\n");      // 错误: 没有全局的printf()
}
```

即使调用的是`std::printf()`，它也还是那个老的C的`printf()` (21.8节)。

注意，这些将使我们可以把具有C连接的库包含到选定的名字空间里，而不去污染全局名字空间。不幸的是，对于将具有C++连接的函数定义在全局名字空间里的头文件，我们就没有同样的灵活手段了。出现此问题，是因为C++实体的连接必须将名字空间考虑在内，因此所生成的目标文件将反应它们在或者不在名字空间里的情况。

9.2.5 连接与指向函数的指针

在一个程序里混用C和C++ 代码片段时，我们有时会希望把在一个语言里定义的函数指针传递到另一个语言中定义的函数里。如果两个语言的两个实现共享同样的连接约定和函数调用机制，这种将指针传递给函数的工作将很简单。但是，一般说没办法去假定这种共性。因此，为了保证一个函数的调用能够符合它被假定的调用方式，我们就必须当心。

在为—一个声明刻画连接约定时，这个特殊约定将应用于由此声明（声明组）引进的所有函数类型、函数名和变量名。这样就可以做出各种奇怪的——有时也是必不可少的——连接可能性的组合。例如，

```
typedef int (*FT) (const void*, const void*); // FT具有C++ 连接

extern "C" {
    typedef int (*CFT) (const void*, const void*); // CFT具有C连接
    void qsort(void* p, size_t n, size_t sz, CFT cmp); // cmp具有C连接
}

void isort(void* p, size_t n, size_t sz, FT cmp); // cmp具有C++连接
void xsort(void* p, size_t n, size_t sz, CFT cmp); // cmp具有C连接
extern "C" void ysort(void* p, size_t n, size_t sz, FT cmp); // cmp具有C++连接

int compare(const void*, const void*); // compare() 具有C++连接
extern "C" int ccmp(const void*, const void*); // ccmp() 具有C连接

void f(char* v, int sz)
{
    qsort(v, sz, 1, &compare); // 错误
    qsort(v, sz, 1, &ccmp);    // ok

    isort(v, sz, 1, &compare); // ok
    isort(v, sz, 1, &ccmp);    // 错误
}
```

如果一个实现中的C和C++ 连接采用同样的调用约定，它就可以接受上面的错误情况，作为对语言的一种扩充。

9.3 使用头文件

为了阐释头文件的各种使用方式，我将展示如何采用几种不同的方式表述计算器程序 (6.1节、8.2节) 的物理结构。

9.3.1 单一头文件

对将一个程序划分成几个文件的最简单解决方案就是将所有定义放入合适数目的 .c 文件

里，在一个头文件里声明这些 .c 文件之间通信所需要的类型，并让它们中的每个都 `#include` 该头文件。对于计算器程序而言，我们可以用5个 .c 文件——`lexer.c`、`parser.c`、`table.c`、`error.c`和`main.c`，用它们保存所有的函数和数据定义；再加上一个`dc.h`，其中存储在不止一个 .c 文件里使用的所有名字的声明。

头文件`dc.h`看起来大致是下面的样子：

```
// dc.h:

namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}

#include <string>

namespace Lexer {
    enum Token_value {
        NAME,          NUMBER,      END,
        PLUS='+',       MINUS='-',    MUL='*',             DIV='/',
        PRINT=';',      ASSIGN='=',  LP='(',              RP=')'
    };

    extern Token_value curr_tok;
    extern double number_value;
    extern std::string string_value;
    Token_value get_token();
}

namespace Parser {
    double prim(bool get); // 处理初等项
    double term(bool get); // 乘和除
    double expr(bool get); // 加和减

    using Lexer::get_token;
    using Lexer::curr_tok;
}

#include <map>

extern std::map<std::string, double> table;

namespace Driver {
    extern int no_of_errors;
    extern std::istream* input;
    void skip();
}
```

将关键字`extern`用在每个变量声明上，可以保证当我们用 `#include` 将 "`dc.h`" 包含到各个 .c 文件后，不会出现重复定义的情况。对应的定义可以在适当的 .c 文件中找到。

把实际代码放到一边，`lexer.c`大致是下面的样子

```
// lexer.c:

#include "dc.h"
#include <iostream>
```

```
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

以这种方式使用头文件，就能保证在头文件中的每个声明都将在某个地方被包含到它的定义所在的文件里。例如，在编译`lexer.c`时，提交给编译器的将是

```
namespace Lexer { // 来自dc.h
    // ...
    Token_value get_token();
}

// ...

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

这就保证了编译器能够查出一个名字的类型描述之间不一致的情况。例如，假设`get_token()`被声明为返回`Token_value`，而定义为返回`int`，对`lexer.c`的编译将因为遇到了类型不匹配的误差而失败。如果缺少某个定义，连接器将捕捉到这个误差。如果缺少某个声明，就会有某个`.c`文件无法编译。

文件`parser.c`看起来像下面这样：

```
// parser.c:

#include "dc.h"

double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }
```

文件`table.c`看起来像下面这样：

```
// table.c:

#include "dc.h"

std::map<std::string, double> table;
```

符号表就是标准库`map`类型的一个变量，这里把`table`定义成了全局变量。在实际规模的程序里，这类对于全局名字空间的小污染也会累积起来，最终造成严重问题。我在这里留下一个瑕疵，就是为了能有机会提出对此问题的警告。

最后，`main.c`将具有下面的样子

```
// main.c:

#include "dc.h"
#include <sstream>

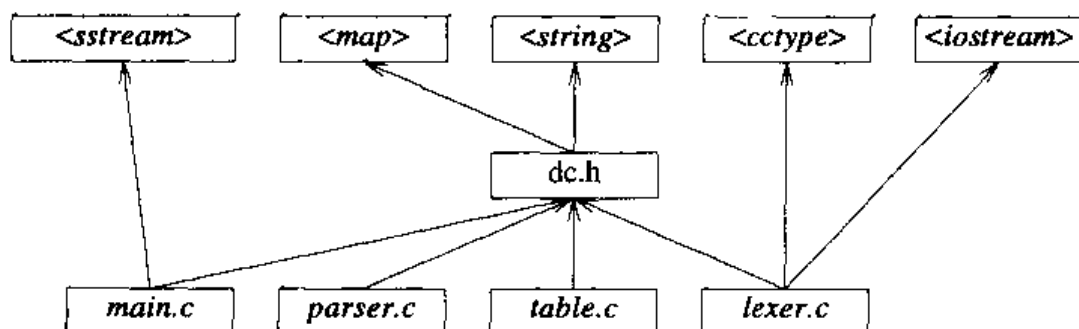
int Driver::no_of_errors = 0;
std::istream* Driver::input = 0;

void Driver::skip() { /* ... */ }

int main(int argc, char* argv[]) { /* ... */ }
```

为能被当做整个程序的`main()`，必须将`main()`声明为全局函数，所以这里没用名字空间。

整个系统的物理结构可以用下面图形表示：



请注意，位于最上面的头文件都是标准库提供的头文件。对于许多程序分析形式而言，这些库都可以忽略不管，因为它们是众所周知的、很稳定的。对于很小的程序，这种结构还可以简化，只要将所有的 `#include` 指令都移到那个公共的头文件里。

当程序很小，而且其中各个部分都不打算分开使用时，采用这种单一头文件风格的物理划分就很合适。请注意，因为这里使用了名字空间，在 `dc.h` 里仍然表现出程序的逻辑结构。如果不使用名字空间，结构就会模糊了，虽然写一些注释可能会有所帮助。

对于更大的程序，在常规的基于文件的开发环境中，单一头文件方式就无法工作了。因为，对公共头文件的修改就将导致对整个程序的重新编译，几个程序员去修改一个头文件也极易造成错误。除非我们进一步把强调的重点放到依靠名字空间和类的程序设计风格上，否则，随着程序的增大，其逻辑结构的品质就会逐渐恶化。

9.3.2 多个头文件

另一种物理组织方式是让每一个逻辑模块有自己的头文件，其中定义它所提供的功能。这样，每个 `.c` 文件用一个对应的 `.h` 文件描述它所提供的东西（它的界面）。每个 `.c` 文件包含它自己的 `.h` 文件，或许还需要包含其他 `.h` 文件，如果那里描述了本 `.c` 文件为实现自己在界面中说明的服务所需要的有关其他文件的信息。这种物理组织对应于模块的逻辑组织，为用户提供提供的界面放在它的 `.h` 文件里，为实现部分所用的界面放在另于以 `_impl.h` 为后缀的文件里，而模块中的函数、变量等的定义则放在 `.c` 文件里。按照这种方式，分析器将表示为三个文件。分析器的用户界面由 `parser.h` 提供

```
// parser.h:
namespace Parser {      // 给用户的界面
    double expr(bool get);
}
```

为实现分析器的函数所共享的环境通过 `parser_impl.h` 提供

```
// parser_impl.h:
#include "parser.h"
#include "error.h"
#include "lexer.h"

namespace Parser {      // 给实现的界面
    double prim(bool get);
    double term(bool get);
}
```



```

double expr(bool get);

using Lexer::get_token;
using Lexer::curr_tok;
}

```

将`parser.h`文件 `#include`进来, 是为了使编译器能检查一致性(9.3.1节)。

实现分析器的函数都存放在`parser.c`里, 用 `#include`指令包含`Parser`的函数所需要的头文件:

```

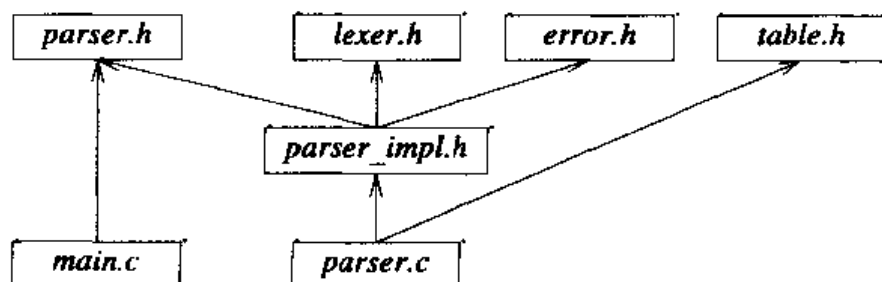
// parser.c:

#include "parser_impl.h"
#include "table.h"

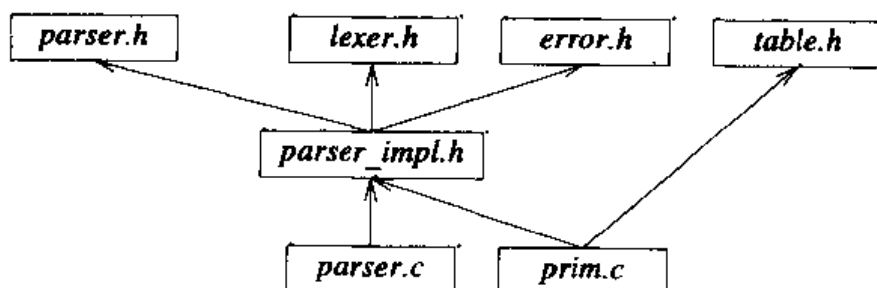
double Parser::prim(bool get) { /* ... */ }
double Parser::term(bool get) { /* ... */ }
double Parser::expr(bool get) { /* ... */ }

```

用图形表示, 分析器和使用它的驱动程序看起来像下面这样



正如我们所希望的, 它非常接近8.3.3节描述的逻辑结构。为了简化这个结构, 我们也可以将 `#include table.h` 放入 `parser_impl.h` 而不是 `parser.c`。无论如何, `table.h` 是这样的一个例子, 在表示分析器函数的共享环境时并不需要它, 只是函数的实现需要它。事实上, 只有一个函数 `prim()` 需要 `table.h`, 所以, 如果我们确实需要维持最小的依赖性, 那么就可以将 `prim()` 放入一个自己的 `.c` 文件, 且只在这里写 `#include table.h`:



除非要处理的模块规模更大, 否则就无须做这种精细加工。即使对真实规模的模块, 用 `#include` 包含一些额外的只是个别函数所需要的文件也很常见。此外, 存在多个 `_impl.h` 的情况也很常见, 因为在一个模块里, 不同的函数子集可能需要不同的共享环境。

请注意, `_impl.h` 的写法既不是标准, 也不是一种通用的约定, 它只不过是其所喜欢的一种命名方式。

为什么要费心于这种多个头文件的复杂模式? 显然, 将所有声明简单地丢进单一头文件里, 费的心思要少得多, 就像前面对 `dc.h` 所做的那样。

多头文件组织方式能够适应于比我们玩具式的分析器大几个数量级的模块，以及比我们的计算器大几个数量级的程序。采用这种组织类型的基本原因，就是它提供了我们所关心的更好的局部性。在分析和修改大程序时，对于程序员而言，最重要的东西就是能将注意力集中于相对较小的代码块。多个头文件的组织方式使人能很容易去确定分析器代码依赖于什么，可以忽略掉程序的其他部分。而单一头文件途径则会强迫我们去关注可能由某个模块使用的所有声明，设法确定它是不是与我们有关的东西。一个很简单的事实是，维护代码的工作总是根据不完全的信息和局部的视角去完成的。多个头文件的组织方式使我们能仅仅从某种局部观察点出发，“自里向外”成功地工作。单一头文件方式——就像所有其他以某种全局性的信息宝库为中心的组织的组织一样——需要一种自上而下的工作方式，从而使我们永远弄不明白到底什么东西依赖于什么东西。

更好的局部化将能减少编译一个模块所需要的信息，从而导致更快的编译。这个影响也可能是非常显著的。我看到过这样的情况，经过简单的依赖性分析而更好地使用头文件，使编译时间减少到只有原来的十分之一。

9.3.2.1 其他计算器模块

计算器的其他模块也可以用类似分析器的方式组织起来。然而，由于这些模块非常小，它们就没有必要再有自己的 `_impl.h` 文件。只有在逻辑模块由许多函数组成，而它们又需要一个共享环境时，才需要使用这种头文件。

错误处理器已经缩减为一组异常类型，根本不需要 `error.c`：

```
// error.h:
namespace Error {
    struct Zero_divide { };

    struct Syntax_error {
        const char* p;
        Syntax_error(const char* q) { p = q; }
    };
}
```

词法处理器提供的是一个相当大的而且比较污浊的界面：

```
// lexer.h:
#include <string>
namespace Lexer {
    enum Token_value {
        NAME,          NUMBER,      END,
        PLUS='+',      MINUS='-',    MUL='*',      DIV='/',
        PRINT=';',     ASSIGN='=',  LP='(',      RP=')'
    };

    extern Token_value curr_tok;
    extern double number_value;
    extern std::string string_value;

    Token_value get_token();
}
```

除了 `lexer.h` 外，词法处理器的实现还依赖于 `error.h`、`<iostream>`，以及在 `<cctype>` 里声明的那些用于确定字符类别的函数：

```
// lexer.c:
#include "lexer.h"
#include "error.h"
#include <iostream>
#include <cctype>

Lexer::Token_value Lexer::curr_tok;
double Lexer::number_value;
std::string Lexer::string_value;

Lexer::Token_value Lexer::get_token() { /* ... */ }
```

我们也可以将对于`error.h`的 `#include`指令提取出来，放入一个`lexer`的 `_impl.h`文件里。当然，我认为对这种小程序而言，那样做实在太过分了。

与平常一样，我们将这个模块提供的界面（在这里是`lexer.h`）用 `#include`包含到模块的实现中，以使编译器能做一致性检查。

符号表基本上是自足的，因为标准库头文件 `<map>` 能够带来与之相关的所有东西，那里实现了--一个有效的`map`模板类：

```
// table.h:
#include <map>
#include <string>

extern std::map<std::string, double> table;
```

由于我们假定每个头文件都可能通过 `#include`包含到几个 `.c`文件里去，我们必须将`table`的声明与其定义分开，虽然`table.c`和`table.h`的差别仅在一个关键字`extern`：

```
// table.c:
#include "table.h"

std::map<std::string, double> table;
```

简而言之，驱动程序依赖于所有的东西：

```
// main.c:
#include "parser.h"
#include "lexer.h"
#include "error.h"
#include "table.h"

namespace Driver {
    int no_of_errors;
    std::istream* input;
    void skip();
}

#include <sstream>

int main(int argc, char* argv[]) { /* ... */ }
```

由于`Driver`名字空间只是由`main()`使用，因此我将它放在`main.c`里。我也可以换另一种方式，将它分出来做成一个`driver.h`，并用 `#include`包含进来。

对于更大的系统，通常值得考虑进一步的组织，使驱动程序直接依赖的东西更少一些。也经常值得将`main()`所做的事情最小化，让`main()`只是去调用一个位于另一个独立源文件

里的驱动函数。对于想作为库去使用的代码，这样做就特别重要。因为此后我们将不能依靠 `main()` 里的代码，而必须为来自各种各样函数的调用做好准备（9.6[8]）。

9.3.2.2 头文件的使用

一个程序里所用的头文件数目会受到许多因素的影响。这些因素中大部分都与你所用的系统里处理文件的方式有关，与C++的关系则小一些。例如，如果你的编辑器没有同时查看许多文件的功能，使用多个头文件就会变得不那么有吸引力。与此类似，打开和读入20个各有50行的头文件也比读入一个1 000行的头文件更耗费时间。你在对小项目采用多头文件风格之前还是应该多考虑一下。

还要给一个警告：十几个头文件再加上程序的执行环境所用的标准头文件（它们常常有上百个）一般还是可以管理的。但是，如果你把一个大程序的声明划分为逻辑上具有最小规模的头文件（将每个结构声明放入它自己独立的头文件，如此等等），对于一个不太大的项目，你很容易就会陷进无法管理的成百个文件的泥沼。我认为那样做就太过分了。

对于大型项目，采用多头文件方式是不可避免的。对于这样的项目，数百个文件（不计标准头文件）是很正常的事情。真正的混乱开始于文件数达到数千时。在那种规模上，这里所讨论的基本技术仍然有用，但对于它们的管理将变成一项极其艰巨的工作。显然，对于真实规模的程序，单一头文件的风格根本不可能使用。这种程序必然有多个头文件。在两种组织风格之间的选择将会（反复地）出现在组成程序的各个部分之中。

单一头文件风格和多头文件风格并不是相互排斥的替代物。它们是具有互补性的两种技术，在设计重要的模块时必须认真考虑选取，在系统演化中也必须重新考虑。最关键的是应该记住，一个界面不可能对于做什么都好。通常很值得将实现方的界面和用户方的界面区分开。此外，许多大系统的结构中都为大部分用户提供了一个简单界面，而为专家用户提供了另一个扩展的界面，这是很好的主意。专家用户的界面（“完整界面”）倾向于用 `#include` 包含比普通用户希望知道的更多的特征。事实上，要弄清普通用户的界面常可以这样做：如果某些特征需要包含一些头文件，其中定义了普通用户不必知道的功能，那么就把它都删掉。术语“普通用户”并没有一点贬义。在那些我不必是专家的领域里，我就特别愿意做一个普通用户，这样我就可以不必去争辩什么了。

9.3.3 包含保护符

多个头文件的想法就是把每个逻辑模块表达为一个统一而自给自足的单位。从整个程序的观点看，为使每个逻辑模块完整的那些声明中将有许多是多余的。对于大型程序，这种冗余也可能带来错误，例如将一个包含类定义或者 `inline` 函数的头文件两次用 `#include` 包含到同一个编译单位里（9.2.3节）。

我们有两种选择。可以：

[1] 重新组织我们的程序。去掉这种冗余。

[2] 找到一种允许重复包含头文件的方式。

对于真实规模的程序而言，采用第一种方式——由它得到计算器的第一个版本——是很麻烦和不实际的。我们还需要一些冗余，以便使程序中分离的各个孤立部分也容易理解。

对多余的 `#include` 的分析和所产生的程序简化有可能带来很多益处，无论是从逻辑的观点，还是编译时间的缩短。然而这种工作却很少可能做完全，所以还需要使用一些能允许多

余的 `#include` 的方法。这种方法最好是能够系统化地使用，因为没有办法知道用户认为有价值的分析能做得多么彻底。

传统的解决方案是在头文件里插入包含保护符。例如，

```
// error.h:
#ifndef CALC_ERROR_H
#define CALC_ERROR_H

namespace Error {
    // ...
}

#endif // CALC_ERROR_H
```

如果 `CALC_ERROR_H` 被定义，位于 `#ifndef` 和 `#endif` 之间的文件内容就会被忽略掉。这样，当 `error.h` 在一次编译中被第一次看到时，它的内容将被读入，而 `CALC_ERROR_H` 也将被给定了一个值。如果在这次编译中 `error.h` 被又一次提供给编译器，它的内容就会被忽略。这是一种黑客式的宏手段，但它很有效，在 C 和 C++ 世界里使用广泛。标准头文件通常也都带有包含保护符。

头文件可能被包含到任意的环境里，而且也没有名字空间保护来抵御宏名字之间的冲突。因此，我总选择相当长的非常难看的名字作为我的包含保护符。

一旦人们习惯于头文件和包含保护符，他们往往就会倾向于直接或间接地包含大量头文件。即使是使用某些优化了头文件处理功能的 C++ 实现，我们也不应该这样做。因为这将导致不必要的过长的编译时间，而且会把大量声明和宏带进作用域中。后一种现象可能以某些无法预计的有害方式影响程序的语义。应该只包含必要的头文件。

9.4 程序

一个程序就是由连接器组合到一起的一组分别编译单位。在这一集合中所使用的每个函数、对象、类型等都必须有惟一的定义（4.9节、9.2.3节）。程序里必须包括恰好一个名字为 `main()` 的函数（3.2节）。程序执行的主要计算将从 `main()` 的调用开始，并由 `main()` 的返回而结束。由 `main()` 返回的 `int` 值被传递给调用 `main()` 的系统，作为程序的结果。

对于包含着全局变量（10.4.9节）或者未被捕捉的异常的程序（14.7节），这个简单的故事还需要进一步加工。

9.4.1 非局部变量的初始化

原则上说，在所有函数之外定义的变量（即那些全局的、名字空间的和类的 `static` 变量）应该在 `main()` 的调用之前完成初始化。在一个编译单位内的这种非局部变量将按照它们的定义顺序进行初始化（10.4.9节）。如果某个这样的变量没有显式的初始式，它将被初始化为有关类型的默认值（10.4.2节）。对于内部类型和枚举，默认的初始值是 0。例如，

```
double x = 2;           // 非局部变量
double y;
double sqx = sqrt(x+y);
```

在这里，`x` 和 `y` 都在 `sqx` 之前初始化，所以调用的是 `sqrt(2)`。

对于不同编译单位里的全局变量，其初始化的顺序则没有任何保证。因此，对于不同编译单位里的全局变量，在它们的初始式之间建立任何顺序依赖都是很不明智的。此外，也没有办法捕捉由全局变量的初始化抛出的异常（14.7节）。一般来说，最好是尽量减少全局变量的使用，特别是限制使用那些要求复杂初始化的全局变量。

也存在着一些技术，可以用于给全局变量的初始化之间强加上某些顺序性。但是，没有一种技术是既可移植又有效的。特别地，动态连接库与具有复杂依赖关系的全局变量也无法很好地共存。

通过函数返回的引用可以作为全局变量的一种很好的替代物。例如，

```
int& use_count()
{
    static int uc = 0;
    return uc;
}
```

对`use_count()`的调用在行为上就像是一个全局变量，除了它是在第一次使用时才被初始化之外（5.5节、7.1.2节）。例如，

```
void f()
{
    cout << ++use_count(); // 读和增加
    // ...
}
```

非局部静态变量的初始化由具体实现中启动C++ 程序所用的机制控制，只有`main()`被执行，这一机制才保证能正确工作。因此，如果要将一段C++ 代码作为某个非C++ 程序的一部分去执行，我们就应该避免在其中定义需要运行中初始化的非局部变量。

请注意，通过常量表达式（C.5节）初始化的变量不会依赖于来自其他编译单位的对象，因此不需要在运行时初始化。这种变量可以安全地用于所有情况。

9.4.1.1 程序终止

一个程序可能以多种方式终止：

- 通过从`main()`返回。
- 通过调用`exit()`。
- 通过调用`abort()`。
- 通过抛出一个未被捕捉的异常。

此外，还有各种各样病态的或者依赖于实现的使程序垮台的方式。

如果一个程序利用标准库函数`exit()`终止，所有已经构造起来的静态对象的析构函数都将被调用（10.4.9节、10.2.4节）。然而，如果程序使用标准库函数`abort()`而终止，那么析构函数就不会被调用。请注意，这意味着`exit()`并不立即终止程序。在析构函数里调用`exit()`有可能导致无穷递归。`exit()`的类型是

```
void exit(int);
```

和`main()`的返回值一样（3.2节），`exit()`的参数也将被作为程序的值返回给“系统”。用零指明程序成功结束。

调用`exit()`结束程序，意味着调用它的函数及其调用者里的局部变量的析构函数都不会执行。抛出一个异常并捕捉它则能保证局部变量被正确地销毁（14.4.7节）。此外，调用`exit()`

将终止程序，不会给调用`exit()`函数的调用者留下处理问题的机会。因此，最好是通过抛出异常以脱离一个环境，让异常处理器去决定下面应该做些什么。

C（和C++）标准库函数`atexit()`使我们可以让程序在终止前执行一些代码。例如，

```
void my_cleanup();

void somewhere()
{
    if (atexit(&my_cleanup) == 0) {
        // 在正常终止时将调用my_cleanup()
    }
    else {
        // 呜呼：atexit函数太多
    }
}
```

这非常像在程序终止时对全局变量自动调用析构函数（10.4.9节、10.2.4节）。请注意，提供给`atexit()`的参数函数不能有参数也不能返回值。另外，这里存在着一个由实现确定的`atexit()`函数的最大数目。`atexit()`通过返回非0值表明已经达到了最大限制。这些约束情况也使`atexit()`不如它初看起来那么有用。

在`atexit(f)`调用之前被静态分配的对象（全局的，10.4.9节；函数`static`，7.1.2节；或者类`static`，10.2.4节）的析构函数将在`f`的调用之后被调用。在一个`atexit(f)`调用之后建立的这种对象的析构函数将在`f`的调用之前被调用。

函数`exit()`、`abort()`和`atexit()`在`<cstdlib>`里声明。

9.5 忠告

- [1] 利用头文件去表示界面和强调逻辑结构；9.1节、9.3.2节。
- [2] 用`#include`将头文件包含到实现有关功能的源文件里；9.3.1节。
- [3] 不要在不同编译单位里定义具有同样名字，意义类似但又不同的全局实体；9.2节。
- [4] 避免在头文件里定义非`inline`函数；9.2.1节。
- [5] 只在全局作用域或名字空间里使用`#include`；9.2.1节。
- [6] 只用`#include`包含完整的定义；9.2.1节。
- [7] 使用包含保护符；9.3.3节。
- [8] 用`#include`将C头文件包含到名字空间里，以避免全局名字；8.2.9.1节、9.2.2节。
- [9] 将头文件做成自给自足的；9.2.3节。
- [10] 区分用户界面和实现界面；9.3.2节。
- [11] 区分一般用户界面和专家用户界面；9.3.2节。
- [12] 在有意向用于非C++ 程序组成部分的代码中，应避免需要运行时初始化的非局部对象；9.4.1节。

9.6 练习

1. (*2) 在你的系统里找到保存标准库头文件的位置。列出它们的名字。是否有其他非标准头文件和它们保存在一起？那些非标准头文件也能用`<>`记法`#include`吗？
2. (*2) 非标准的“基础”库头文件保存在哪里？

3. (*2.5) 写一个程序，它读入一个源文件，并写出其中用 `#include` 包含的所有文件的名字。采用缩排形式显示出被包含的文件里用 `#include` 包含的文件。用某些实际的源程序做这个程序的试验（以便取得对被包含的信息的认识）。
4. (*3) 修改上面练习中的程序，对于每个被包含的文件，打印出其中注释的行数，非注释的行数，非注释的以空白分隔的单词数。
5. (*2.5) 一个外部包含保护符是一种结构，它在文件的外部检查被它保护的代码，并在一次编译中只包含这个文件一次。请设计出这样一种结构，找出一种测试它的方法，将它与第 9.3.3 节所描述的包含保护符相比较，讨论其优点和缺点。外部保护符在你的系统上有什么显著的运行优势吗？
6. (*3) 在你的系统中如何得到动态连接？对于动态连接的代码有什么限制？要使代码成为动态连接的，对于它有什么特别的要求吗？
7. (*3) 打开并读入 100 个各包含 1 500 个字符的文件。打开并读入一个包含 150 000 个字符的文件。提示：参见 21.5.1 节的例子。在执行性能方面有什么差异吗？在你的系统上能够同时打开的最大文件数是多少？针对 `#include` 文件的使用考虑这些问题。
8. (*2) 修改计算器程序，使它可以从 `main()` 或者其他函数里通过一个简单的函数调用而激活。
9. (*2) 对于使用 `error()` 而不是异常的计算器版本（8.2.2 节），画出其“模块依赖关系图”（9.3.2 节）。

第二部分 抽象机制

这一部分描述C++ 定义和使用新类型的功能，介绍通常称为面向对象程序设计和通用型程序设计的技术。

章目

- 第10章 类
- 第11章 运算符重载
- 第12章 派生类
- 第13章 模板
- 第14章 异常处理
- 第15章 类层次结构

没有任何事情比建立新秩序更困难，成功的希望更渺茫，处理起来更危险。因为改革者将使旧秩序的所有既得利益者都变成自己的死敌，但只能使新秩序的可能获益者成为其半心半意的辩护士。

——尼科罗·马基雅维利
(《君主论》(*The Prince*), §vi)

第10章 类

这些类型并不“抽象”；
它们像int和float一样实际
——Doug McIlroy

概念和类——类成员——访问控制——构造函数——*static*成员——默认复制——*const*成员函数——*this*——*struct*——在类内部的函数定义——具体类——成员函数和协助函数——重载运算符——具体类的使用——析构函数——默认构造函数——局部变量——用户定义的复制——*new*和*delete*——成员对象——数组——静态存储——临时变量——联合——忠告——练习

10.1 引言

C++ 里类概念的目标就是为程序员提供一种建立新类型的工具，使这些新类型的使用能够像内部类型一样方便。此外，派生类（第12章）和模板（第13章）提供了一些将相关的类组织起来的方法，使程序员可以利用类之间的相互关系。

一个类型是某个概念的一个具体体现。例如，C++ 内部类型*float*及其运算符 $+$ 、 $-$ 、 $*$ 等，所提供的就是数学中实数概念的一个具体近似。一个类就是一个用户定义类型。我们设计一个新类型，是为了给某个在内部类型中没有直接对应物的概念提供一个定义。举例来说，我们可能在某个处理电话的程序里提供一个类型*Thunk_line*，在某个电子游戏中提供类型*Explosion*，为文字处理程序建立类型*list<Paragraph>*。如果一个程序里提供的类型与应用中的概念有直接的对应，与不具有这种情况的程序相比，这个程序很可能就更容易理解，也更容易修改。一组经过很好选择的用户定义类型可能使一个程序更简洁。此外，它还能使各种形式的代码分析更容易进行。特别地，它还会使编译器有可能检查对象的非法使用。否则，这些错误就很可能遗留下来，一直留到对程序进行彻底测试时。

定义新类型的基本思想就是将实现中那些并非必然的细节（例如，用于存储该类型的对象所采用的数据的布局），与那些对这个类型的正确使用至关重要的性质（例如，能够访问其中数据的完整的函数列表）区分开来。表达这种划分的最好方式就是提供一个特定的界面，令对于数据结构以及内部维护例程的所有使用都通过这个界面进行。

本章将集中关注相对简单的“具体”用户定义类型，它们在逻辑上与内部类型没有多大差异。最理想的情况是，在使用方式上应该无法区分出这种类型与内部类型，使它们的差异仅仅在创建的方式上。

10.2 类

一个类就是一个用户定义类型。本节将介绍定义类、创建类的对象、以及操作这些对象

的基本功能。

10.2.1 成员函数

现在考虑用一个`struct`实现日期的概念，方式是定义一个`Date`的表示和一组对这种类型的数据进行操作的函数

```
struct Date {           // 表示
    int d, m, y;
};

void init_date(Date& d, int, int, int);    // 初始化d
void add_year(Date& d, int n);            // d加n年
void add_month(Date& d, int n);           // d加n月
void add_day(Date& d, int n);             // d加n天
```

在日期类型与这些函数之间并没有任何显式的联系。我们可以通过将这些函数声明为成员，从而建立起这种联系：

```
struct Date {
    int d, m, y;

    void init(int dd, int mm, int yy);    // 初始化
    void add_year(int n);                // 加n年
    void add_month(int n);               // 加n月
    void add_day(int n);                 // 加n天
};
```

在一个类（`struct`也是一种类，10.2.8节）里声明的函数被称做成员函数，这种函数只能通过适当类型的特定变量，采用标准的结构成员访问语法形式调用。例如，

```
Date my_birthday;

void f()
{
    Date today;
    today.init(16, 10, 1996);
    my_birthday.init(30, 12, 1950);

    Date tomorrow = today;
    tomorrow.add_day(1);
    // ...
}
```

由于在不同的结构里有可能存在具有同样名字的成员函数，我们在定义成员函数时必须给出有关结构的名字：

```
void Date::init(int dd, int mm, int yy)
{
    d = dd;
    m = mm;
    y = yy;
}
```

在成员函数里面，结构内部各个成员的名字都可以直接使用，不必显式地去引用某个对象。在这种情况下，这些名字所引用的就是该函数调用时所针对的那个对象的成员。例如，如果为`today`调用`Date::init()`，那么`m = mm`将给`today.m`赋值。另一方面，如果为`my_birthday`调用

`Date::init()`，那么`m = mm`就会给`my_birthday.m`赋值。类的成员函数总知道它是为哪个对象而调用的。

结构

```
class X { ... };
```

被称为一个类定义，因为它定义了一个新类型。由于历史的原因，一个类定义也常常被说成是一个类声明。另外，就像其他不是定义的声明一样，类定义可以由于 `#include` 的使用而在不同的源文件里重复出现，这样并不违反惟一性定义规则（9.2.3节）。

10.2.2 访问控制

在前一节里的 `Date` 声明提供了一组操作 `Date` 的函数，然而，它却没有清楚地说明这些函数就是直接依赖于 `Date` 的表示的全部函数，而且也是仅有的能够直接访问 `Date` 类的对象的函数。这些限制可以通过用 `class` 代替 `struct` 而描述清楚：

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy); // 初始化
    void add_year(int n);             // 加n年
    void add_month(int n);            // 加n月
    void add_day(int n);              // 加n天
};
```

标号 `public` 将这个类的体分为两个部分，其中的第一部分（私用部分）只能由成员函数使用；而第二部分（公用部分）则构成了该类的对象的公用界面。一个 `struct` 也是一个 `class`，但是其成员的默认方式是公用的（10.2.8节）。成员函数的定义和使用与以前完全一样。例如，

```
inline void Date::add_year(int n)
{
    y += n;
}
```

然而，非成员函数将禁止访问私用成员，例如，

```
void timewarp(Date& d)
{
    d.y -= 200; // 错误：Date::y为私用成员
}
```

将对数据结构的访问严格限制到明确说明的一组函数，这样做有许多优点。例如，导致 `Date` 保存某个非法值（例如，1985年12月36日）的任何错误必然是由某个成员函数的代码造成的。这就意味着第一个层次的排错——局部化——在程序运行之前就已经完成了^①。这也是一种一般性认识的特殊情况，该认识是：类型 `Date` 行为的任何改变都能够而且必然是受到其成员函数改变的影响。特别是，如果一个类的表示方式改变了，我们只要修改成员函数，就可以去使用这种新表示。用户代码只直接依赖于公用界面，因此就不需要重新写（虽然可能需要重新编译）。另一个优点是，一个潜在的用户要学习使用一个类，也只需要考察其成员函数

① 这里特别指所有从类外直接访问私用成员的非局部访问，它们都将被编译器作为非法访问而检查出来，完全不必经过程序运行和调试。——译者注

的定义。

对私用成员的保护依靠的是对于使用类成员名字的限制。可以通过地址操作和显式的类型转换等绕过这些限制。不过，这些方式当然是作弊行为。C++ 的保护只是为了避免无意的失误，而不是为了防止精心策划的计谋（骗局）。只有硬件才可能防止对通用语言的恶意使用。当然，即使是利用了硬件，想在现实的系统里把这件事情做好也非常困难。

这里加入进一个 `init()` 函数，部分原因是，有一个函数来设置对象的值一般总是有用的；还有就是数据的私用性也迫使我们去提供它。

10.2.3 构造函数

采用 `init()` 之类的函数提供对类对象的初始化，这样做既不优美又容易出错。因为没有任何地方说一个对象必须经过初始化，程序员有可能忘记去做这件事情——或者做了两次（常常会带来同样具有灾难性的后果）。一种更好的途径是让程序员有能力去声明一个函数，其明确目的就是去完成对象的初始化。因为这样一个函数将构造起一个给定类型的值，它就被称为构造函数。构造函数很容易辨认，它具有与类同样的名字。例如，

```
class Date {
    // ...
    Date(int, int, int);    // 构造函数
};
```

如果一个类有一个构造函数，这个类的所有对象的初始化都将通过对某个构造函数的调用而完成进行初始化（10.4.3节）。如果该构造函数要求参数，那么就必须提供这些参数：

```
Date today = Date(23, 6, 1983);
Date xmas(25, 12, 1990);    // 简写形式
Date my_birthday;          // 错误：缺少初始式
Date release1_0(10, 12);    // 错误：缺少第3个参数
```

允许以多种方式初始化类对象常常也很好。这种事情可以通过提供多个构造函数的方式完成。例如，

```
class Date {
    int d, m, y;
public:
    // ...
    Date(int, int, int);    // 日，月，年
    Date(int, int);        // 日，月，本年
    Date(int);             // 日，本月本年
    Date();                // 默认日期：今天
    Date(const char*);      // 字符串表示的日期
};
```

构造函数也服从与其他函数完全一样的重载规则（7.4节）。只要构造函数在参数类型上存在足够大的差异，编译器就能为每个使用选出一个正确的：

```
Date today(4);
Date july4("July 4, 1983");
Date guy("5 Nov");
Date now;    // 默认地初始化为今天
```

像 `Date` 类里这种的构造函数的多样性非常典型。在设计一个类的时候，程序员总是倾向于去

增加一些他认为某些人可能需要的特征。要确定哪些特征是必需的，且只把它们加进来，反而需要做更多的思考。然而，这种额外的思考通常能带来更小更容易理解的程序。减少相关函数数目的一种方式是采用默认参数（7.5节）。对于**Date**，可以给每个参数一个默认值，并将它们解释为“取默认值：*today*”：

```
class Date {
    int d, m, y;
public:
    Date(int dd=0, int mm=0, int yy=0);
    // ...
};

Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;

    // 检查Date是否合法
}
```

当我们需要用一个参数值来指明“取默认值”时，所选的这个值就必须处于该参数的所有可能值之外。对于*day*和*month*，上述选择很清楚。而对于*year*，0值可能就不是一个明显的选择。幸运的是，在公历中没有0年，公元前1年（*year* == -1）之后就是公元1年（*year* == 1）。

10.2.4 静态成员

为**Date**提供默认值的方便方式实际上隐含着一个重要问题：**Date**类现在依靠着一个全局变量*today*。这样，这个**Date**类就只能在定义了*today*，并且所有代码都正确地使用*today*的环境中使用。这就形成了一种限制，使这个类在写它的那个环境之外基本上就没有用了。如果用户试图去使用这种依赖于环境的类，他们必定会遇到许多很不愉快的事情。维护也会变成麻烦。或许“只有一个小小的全局变量”还不是特别难管理的问题，但是以这种风格产生出的代码，除去写它的程序员之外，基本上就毫无用处。这种情况应该避免。

还好，我们确实有可能获得上面这样的方便，而同时又无须受到访问全局变量之累。如果一个变量是类的一部分，但却不是该类的各个对象的一部分，它就被称为是一个**static**静态成员。一个**static**成员只有惟一的一份副本，而不像常规的非**static**成员那样在每个对象里各有一份副本（C.9节）。与此类似，一个需要访问类成员，然而却并不需要针对特定对象去调用的函数，也被称为一个**static**成员函数。

下面是一次重新设计，其中保留了对于**Date**的默认构造函数值的语义，同时消除了由于依赖全局量而引起的问题：

```
class Date {
    int d, m, y;
    static Date default_date;
public:
    Date(int dd=0, int mm=0, int yy=0);
    // ...
    static void set_default(int, int, int);
};
```

现在我们可以给出**Date**构造函数的如下定义

```
Date::Date(int dd, int mm, int yy)
{
    d = dd ? dd : default_date.d;
    m = mm ? mm : default_date.m;
    y = yy ? yy : default_date.y;
    // 检查Date的合法性
}
```

如果需要的话，我们也可以改变默认日期。静态成员也可以像任何其他成员一样引用。此外，对于静态成员的引用不必提到任何对象，相反，在这里应该给成员的名字加上了作为限定词类名字。例如，

```
void f()
{
    Date::set_default(4, 5, 1945);
}
```

静态成员——包括函数和数据成员——都必须在某个地方另行定义。例如，

```
Date Date::default_date(16, 12, 1770);
void Date::set_default(int d, int m, int y)
{
    Date::default_date = Date(d, m, y);
}
```

这样就将默认值定义为贝多芬的生日了——直到某个人做出另外的决定。

请注意，**Date()**也就成了对**Date::default_date**值的一种记述形式。例如，

```
Date copy_of_default_date = Date();
```

因此，我们就不必为读出默认日期提供专门的函数了。

10.2.5 类对象的复制

按照默认约定，类对象可以复制。特别是可以用同一个类的对象的复制对该类的其他对象进行初始化。即使是声明了构造函数的地方，也还是可以这样做：

```
Date d = today;    // 通过复制初始化
```

按照默认方式，类对象的复制就是其中各个成员的复制。如果某个类**X**所需要的不是这种默认方式，那么就可以定义一个复制构造函数**X::X(const X&)**，由它提供所需要的行为。这个问题将在10.4.4.1节进一步讨论。

类似地，类成员也可以通过赋值进行按默认方式的复制，例如，

```
void f(Date& d)
{
    d = today;
}
```

在这里也一样，默认的语义就是按成员复制。如果对于某个类**X**而言这种方式不是正确选择，用户也能定义合适的赋值运算符（10.4.4.1节）。

10.2.6 常量成员函数

到目前为止，所定义的*Date*只提供了一些给定*Date*值或者改变其值的函数。但是我们还没有提供一些方法来检查一个*Date*的值。这个问题很容易弥补，只要增加几个读出年月日的函数就可以了：

```
class Date {
    int d, m, y;
public:
    int day() const { return d; }
    int month() const { return m; }
    int year() const;
    // ...
};
```

请注意，在函数声明的（空）参数表后面出现的*const*，它指明这些函数不会修改*Date*的状态。很自然，编译器将能捕捉到无意中违背这种承诺的任何企图。例如，

```
inline int Date::year() const
{
    return y++;    // 错误：在const函数里企图修改成员值
}
```

如果在一个类的外面定义它的*const*成员函数，也需要给出*const*后缀：

```
inline int Date::year() const    // 正确
{
    return y;
}

inline int Date::year()    // 错误：在成员函数类型中缺少const
{
    return y;
}
```

换句话说，*const*也是*Date::day()*和*Date::year()*的类型中的一部分。

对于*const*或者非*const*对象都可以调用*const*成员函数，而非*const*成员函数则只能对非*const*对象调用。例如，

```
void f(Date& d, const Date& cd)
{
    int i = d.year();    // ok
    d.add_year(1);    // ok

    int j = cd.year();    // ok
    cd.add_year(1);    // 错误：不能修改const cd的值
}
```

10.2.7 自引用

状态更新函数*add_year()*、*add_month()*和*add_day()*都被定义为不返回值的函数。对于这样一组相关的更新函数，可以让它们返回一个到被更新对象的引用，以使对于对象的操作可以串接起来。这也是一种很有用的技术。例如，我们可能希望写

```
void f(Date& d)
{
```

```

// ...
d.add_day(1).add_month(1).add_year(1);
// ...
}

```

给`d`加上一年一个月零一天。为此，每个函数都应声明为返回一个到`Date`的引用值：

```

class Date {
// ...
Date& add_year(int n); // 加n年
Date& add_month(int n); // 加n月
Date& add_day(int n); // 加n天
};

```

每个（非静态）函数都知道它是为了哪个对象而调用的，因此可以显式地引用该对象。例如，

```

Date& Date::add_year(int n)
{
    if (d==29 && m==2 && !leapyear(y+n)) { // 当心2月29日
        d = 1;
        m = 3;
    }
    y += n;
    return *this;
}

```

表达式 `*this` 引用的就是这个函数的这次调用所针对的那个对象。它等价于 *Simula* 的 `THIS` 和 *Smalltalk* 的 `self`。

在一个非静态的成员函数里，关键字 `this` 是一个指针，指向该函数的当时这次调用所针对的那个对象。在类 `X` 的非 `const` 成员函数里，`this` 的类型就是 `X*`。然而，`this` 并不是一个常规变量，不能取得 `this` 的地址或者给它赋值。在类 `X` 的 `const` 成员函数里，`this` 的类型是 `const X*`，以防止对于这个对象本身的修改（5.4.1 节）。

大部分对于 `this` 的应用都是隐含的。特别是，对一个类中非静态成员的引用都要依靠隐式地使用 `this`，以获取相应对象的成员。例如，函数 `add_year` 可以（虽然很麻烦）等价地修改为下面的样子：

```

Date& Date::add_year(int n)
{
    if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
        this->d = 1;
        this->m = 3;
    }
    this->y += n;
    return *this;
}

```

`this` 的一种常见的显式使用是在链接表的操作中（例如，24.3.7.4 节）。

10.2.7.1 物理的和逻辑的常量性

偶然有这种情况，一个成员函数在逻辑上是 `const`，但它却仍然需要改变某个成员的值。对于用户而言，这个函数看似没有改变其对象的状态，然而，它却可能更新了某些用户不能直接访问的细节。这通常被称为逻辑的常量性。例如，`Date` 类可能有一个函数，它应返回一个用户可以用于打印的字符串表示。构造出这种表示可能是一个相对费时的操作，因此，保

留一个副本，在重复需要的时候直接返回这个副本，这一做法也就有意义了，除非这个**Date**值被改变。对于更复杂的数据结构，缓存起这种值的情况很常见。现在让我们来看如何对**Date**达到这种效果：

```
class Date {
    bool cache_valid;
    string cache;
    void compute_cache_value(); // 填充缓存
    // ...
public:
    // ...
    string string_rep() const; // 字符串表示
};
```

从用户的角度看，**string_rep**并没有改变它的**Date**的状态，所以它应该是个**const**成员函数。而在另一方面，在使用之前必须填充缓存，要做到这一点只能通过蛮力：

```
string Date::string_rep() const
{
    if (cache_valid == false) {
        Date* th = const_cast<Date*>(this); // 强制去掉const
        th->compute_cache_value();
        th->cache_valid = true;
    }
    return cache;
}
```

也就是说，这里用**const_cast**运算符（15.4.2.1节）从**this**获得一个**Date***指针。这当然一点也不优美，而且它也无法保证总能工作，例如，当被应用的对象原本就是一个**const**的时候。看例子

```
Date d1;
const Date d2;
string s1 = d1.string_rep();
string s2 = d2.string_rep(); // 无定义行为
```

对于**d1**的情况，**string_rep()**简单地将其强制转回原来的类型，所以这个调用能够完成。然而，**d2**原本定义为**const**，具体实现很有可能为保护它的值不被破坏而使用的某种特殊形式的存储。因此，无法保证**d2.string_rep()**能在所有实现上都给出同样的可预见的结果。

10.2.7.2 可变的——*mutable*

显式类型转换“强制去掉**const**”，以及由它引起的依赖于实现的行为还是可以避免的，只要将缓存管理所涉及的数据声明为**mutable**：

```
class Date {
    mutable bool cache_valid;
    mutable string cache;
    void compute_cache_value() const; // 填充 (mutable) 缓存
    // ...
public:
    // ...
    string string_rep() const; // 字符串表示
};
```

存储描述符`mutable`特别说明这个成员需要以一种能允许更新的方式存储——即使它是某个`const`对象的成员。换言之，`mutable`意味着“不可能是`const`”。这种机制可用于简化`string_rep()`的定义

```
string Date::string_rep() const
{
    if (!cache_valid) {
        compute_cache_value();
        cache_valid = true;
    }
    return cache;
}
```

这就使`string_rep()`的合理使用都能合法化了。例如，

```
Date d3;
const Date d4;
string s3 = d3.string_rep();
string s4 = d4.string_rep();    // ok!
```

如果在某个表示中（只有）一部分允许改变，将这些成员声明为`mutable`是最合适的。如果在一个对象在逻辑上保持为`const`的同时，其中的大部分都需要修改，那么最好将这些需要修改的数据放入另一个独立的对象里，并间接地访问它。假设采用这种技术，使用缓存的字符串将变成

```
struct cache {
    bool valid;
    string rep;
};

class Date {
    cache* c;                // 在构造函数里初始化（10.4.6节）
    void compute_cache_value() const; // 填充引用的缓存
    // ...
public:
    // ...
    string string_rep() const;    // 字符串表示
};

string Date::string_rep() const
{
    if (!c->valid) {
        compute_cache_value();
        c->valid = true;
    }
    return c->rep;
}
```

这种支持缓存的技术可以推广到各种形式的延迟求值（lazy evaluation）。

10.2.8 结构和类

按照定义，一个`struct`也就是一个类，但其成员默认为公用的；也就是说，

```
struct s { ...
```

只是下面定义的简写形式

```
class s { public: ...
```

可以用访问描述符`private`: 说明紧随其后的一些成员都是私用的, 就像`public`: 说明随它之后的成员是公用的一样。除了名字不同之外, 下面两个声明完全等价:

```
class Date1 {
    int d, m, y;
public:
    Date1(int dd, int mm, int yy);

    void add_year(int n);    // 加n年
};

struct Date2 {
private:
    int d, m, y;
public:
    Date2(int dd, int mm, int yy);

    void add_year(int n);    // 加n年
};
```

采用哪种形式要看情况和你自己的偏好。我通常将`struct`用于所有成员都是公用的那些类。我认为这样的类“并不是完整的类型, 不过是个数据结构”。对于这种结构, 构造函数和访问函数也是相当有用的, 但只是作为一种简写形式, 而不是作为类型性质(不变式, 24.3.7.1节)的捍卫者。

并不要求一定将数据放在类的最前面声明。事实上, 将数据放到最后, 以突出函数所提供的用户界面, 也是一种很合理的方式。例如,

```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
    void add_year(int n);    // 加n年
private:
    int d, m, y;
};
```

在实际的代码里, 那里的公用界面和实现细节都远比在教学实例中多得多, 这时我通常更喜欢`Date3`的风格。

访问描述符可以在一个类声明中多次使用。例如,

```
class Date4 {
public:
    Date4(int dd, int mm, int yy);
private:
    int d, m, y;
public:
    void add_year(int n);    // 加n年
};
```

像`Date4`那样出现多个公用段, 也容易使代码趋向混乱。具有多个私用段也一样。但无论如何, 如果用机器生成代码, 在一个类里允许多个访问描述符就很有价值了。

10.2.9 在类内部的函数定义

如果一个函数是在类定义的内部定义的——而不是仅仅在这里声明——它就被作为一个内联成员函数。也就是说，在类内部定义的成员函数应该是小的、频繁使用的函数。在类内部定义的成员函数与它们所在的类定义一起，也可能因为 `#include` 的使用而在某些编译单位里重复出现。与类本身一样，这种函数的意义在所有使用中都必须一致（9.2.3节）。

采用将数据成员放在最后的风格，有可能给需要访问这些数据表示的公用 `inline` 函数带来一点小问题。例如，

```
class Date {    // 可能使人糊涂
public:
    int day() const { return d; }    // 返回Date::d
    // ...
private:
    int d, m, y;
};
```

这是完全正确的C++ 代码，因为在一个类里声明的成员函数可以访问该类的每个成员，就像在考虑这个成员函数的体时，该类已经完全定义好了一样。当然，这种情况也可能把读程序的人弄糊涂。

正因为如此，我通常还是把数据放在前面，或者是把 `inline` 成员函数的定义紧随在类定义之后，例如，

```
class Date {
public:
    int day() const;
    // ...
private:
    int d, m, y;
};

inline int Date::day() const { return d; }
```

10.3 高效的用户定义类型

前一节在介绍语言中有关定义类的基本特征的大环境下，讨论了设计一个 `Date` 类的种种细节。现在我要回过头来，强调和讨论设计一个简单而高效的 `Date` 类，并阐明语言的各种特征将能怎样支持这种设计。

较小的使用极其频繁的抽象在许多应用系统中都很常见。这样的例子如拉丁字母、中文字、整数、浮点数、复数、点、指针、坐标、变换、(指针, 偏移量) 对偶、日期、时间、范围、链接、关联、结点、(值, 单位) 对偶、磁盘地址、源代码位置、BCD 字符、现金、线段、矩形、规范化的浮点数、带分数、字符串、向量、数组等。每个应用都会用到这其中的某些东西。常见情况是这些简单类型中的某几个被极频繁地使用，一个典型应用里将直接用到其中的几个，间接地通过库使用的则更多。

C++ 和其他程序设计语言直接支持其中的若干种抽象。当然，对大部分都不直接支持，而且也无法支持，因为这种东西实在太多了。进一步说，通用程序设计语言的设计者也不可能预见到每一个应用的细节。因此，语言中就必须提供一些机制，使用户可以定义这些小的

具体类型。将这些类型称做具体类型或者具体类，是为了将它们与抽象类（12.3节）和类层次结构（12.2.4节、12.4节）中的类相区别。

C++ 的一个明显目标就是要支持这种定义，还要支持这种用户定义数据类型的有效使用。这些都是优美的程序设计的基础。与通常情况一样，简单而世俗的东西，从统计上看，远比那些复杂而精妙的东西重要得多。

根据这种看法，让我们来构筑一个更好的日期类：

```
class Date {
public:    // 公用界面:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    class Bad_date {}; // 异常类

    Date(int dd=0, Month mm=Month(0), int yy=0); // 0的意思是“取默认值”

    // 检查Date的函数:
    int day() const;
    Month month() const;
    int year() const;
    string string_rep() const;    // 字符串表示
    void char_rep(char s[]) const; // C风格的字符串表示

    static void set_default(int, Month, int);

    // 修改Date的函数:
    Date& add_year(int n);    // 加n年
    Date& add_month(int n);    // 加n月
    Date& add_day(int n);    // 加n天
private:
    int d, m, y;    // 表示
    static Date default_date;
};
```

对于用户定义类型，这样的一组操作很典型：

- [1] 一个构造函数描述这个类型的对象/变量应该如何初始化。
- [2] 一组函数使用户可以查看**Date**。这些函数标记为**const**，表示在对对象/变量调用时，这些函数不会修改对象的状态。
- [3] 一组函数使用户可以操作**Date**，而又不必知道其细节表示，也不必去直接摆弄其中复杂的语义。
- [4] 一组隐式定义的函数，使**Date**可以自由地复制。
- [5] 一个类，**Bad_date**，用于通过异常报告出错情况。

我还定义了一个**Month**类型，这是为了对付记忆的问题，例如，6月7日是写成**Date(6, 7)**（按美国风格）还是**Date(7, 6)**（按欧洲风格）。我还增加了一种机制去处理默认参数。

我也考虑了引进独立的类型**Day**和**Year**，以便能处理可能的混乱情况**Date(1995, jul, 27)**和**Date(27, jul, 1995)**。然而这些类型并不像类型**Month**那么有用，因为几乎所有错误都能在运行时捕捉到——27年的7月26日在我的工作中并不是常见的日期。如何处理1800年之前的历史日期之类的事情过于技术性，最好还是留给历史学的专家^①。进一步说，一个月的日数

① 由于历法等方面的情况和一些错误，西方历史的记载中存在着非常复杂的日期问题。因此作者有这里的说法。——译者注

在脱离了它所在的年和月之后也无法正确地检查。有关定义一个方便类型`Year`的问题可参见11.7.1节。

默认日期必须在某个地方定义为一个合法的日期。例如，

```
Date Date::default_date(22,jan,1901);
```

我将忽略10.2.7.1节中提出的缓存技术，因为对一个简单类型而言并没有这种必要。如果真的需要，也可以将它作为一个实现细节加进来，对用户界面不会产生任何影响。

下面是一个小的——但也是挖空心思的——例子，说明`Date`能够如何使用：

```
void f(Date& d)
{
    Date lvb_day = Date(16, Date::dec, d.year());
    if (d.day() == 29 && d.month() == Date::feb) {
        // ...
    }
    if (midnight()) d.add_day(1);
    cout << "day after: " << d+1 << '\n';
}
```

这里假定已经声明了针对`Date`类型的输出运算符 `<<` 和加运算符 `+`。我将在10.3.3节里完成这些事情。

请注意`Date::feb`的记法。函数`f`并不是`Date`的成员，因此它必须特别说明要用的是`Date`的`feb`，而不是其他实体。

为什么要为简单到像日期这样的东西定义一个特殊的类型呢？毕竟我们可以定义一个结构

```
struct Date {
    int day, month, year;
};
```

并让程序员去决定如何去用它。但是，如果我们真的那样做了，那么每个用户或者是不得不自己去直接操作`Date`的成分，或者需要提供某些函数去做这些事情。从影响上看，`Date`这个概念将会散布到整个系统里，这将会使程序更难理解，更难做出文档，更难修改。一个无法回避的情况是，用一个简单的结构来提供一个概念，将使这个结构的每个用户都做许多额外的工作。

另外，虽然`Date`类型看似简单，要将它做正确也需要一些思考。例如，增加一个`Date`就必须处理闰年问题，还有不同月份的不同长度，如此等等（注意：10.6[1]）。再说，这种年月日的表示对于许多应用也并不合适。现在如果我们决定改变，我们就只需要修改非常确定的一组函数。例如，要将`Date`表示为1970年1月1日之前或者之后的天数，我们就只要修改`Date`的成员函数（10.6[2]）。

10.3.1 成员函数

很自然，对每个成员函数，都必须在某个地方有一个实现。例如，这里是`Date`的构造函数的定义：


```

Date::Date(int dd, Month mm, int yy)
{
    if (yy == 0) yy = default_date.year();
    if (mm == 0) mm = default_date.month();
    if (dd == 0) dd = default_date.day();

    int max;

    switch (mm) {
    case feb:
        max = 28+leapyear(yy);
        break;
    case apr: case jun: case sep: case nov:
        max = 30;
        break;
    case jan: case mar: case may: case jul: case aug: case oct: case dec:
        max = 31;
        break;
    default:
        throw Bad_date(); // 有人捣乱
    }

    if (dd < 1 || max < dd) throw Bad_date();

    y = yy;
    m = mm;
    d = dd;
}

```

构造函数检查送给它的数据是否表示了一个合法 *Date*，如果不是，例如给的是 *Date(30, Date::feb, 1994)*，它就抛出一个异常（8.3节、第14章），指出有什么事情以某种无法忽视的方式出了毛病。如果提供的数据可以接受，它就完成明确的初始化工作。初始化是相对比较复杂的操作，因为它们往往涉及到对数据合法性检查。在另一方面，一个 *Date* 一旦被建立，它就可以使用和赋值，而无须任何检查了。换句话说，构造函数需要建立起这个类的不变式（在这里就是合法的日期）。其他函数可以依靠这个不变式，然而又必须维护它。这一设计技术将能极大地简化代码（24.3.7.1节）。

我用值 *Month(0)*——它不表示任何月份值——表示“取默认月份值”。我也可以开始时就在 *Month* 里定义一个特别的枚举符来代表这件事情。但是我觉得用一个明显的匿名值表示“取默认月份值”更好一些，还是不要让一年看起来像有13个月。请注意，0可以在这里用，也因为能保证它总是位于枚举 *Month* 的范围之内（4.8节）。

我考虑过将数据的合法性检查提取出来，放进一个单独的函数 *is_date()*。但是，我发现这样做的结果将使用户代码更加复杂，而且不如依赖于捕捉异常的代码那么健壮。例如，假定 >> 已经对 *Date* 有了定义：

```

void fill(vector<Date>& aa)
{
    while (cin) {
        Date d;
        try {
            cin >> d;
        }

        catch (Date::Bad_date) {

```

```

        // 我的错误处理
        continue;
    }
    aa.push_back(d); // 参见3.7.3节
}
}

```

就像对于简单具体类型常见的情况一样，成员函数的定义变化范围很大，从极平凡的，到相当复杂的。例如，

```

inline int Date::day() const
{
    return d;
}

Date& Date::add_month(int n)
{
    if (n==0) return *this;
    if (n>0) {
        int delta_y = n/12;
        int mm = m+n%12;
        if (12 < mm) { // 注意: int(dec) == 12
            delta_y++;
            mm -= 12;
        }

        // 处理Month(mm)没有天数d的情况

        y += delta_y;
        m = Month(mm);
        return *this;
    }

    // 处理负数n

    return *this;
}

```

10.3.2 协助函数

经常有这种情况，一个类有一批与它相关联的函数，而它们又未必要定义在类里，因为它们并不需要直接访问有关的表示。例如，

```

int diff(Date a, Date b); // 在范围 [a, b) 或者 [b, a) 中的天数
bool leapyear(int y);
Date next_weekday(Date d);
Date next_saturday(Date d);

```

在类中定义这些函数将使类的界而复杂化，也增加了在改变表示时需要检查的函数个数。

这些函数怎样与类**Date**“相关联”呢？按照传统方式，它们的声明将直接与类**Date**的声明放在同一个文件里，那些需要**Date**的用户在包含了定义它的界面文件时（9.2.1节），也就使这些函数都可以使用了。例如，

```
#include "Date.h"
```

除了使用一个特殊的**Date.h**头文件之外，或者说是作为另一种替代方式，我们还可以将这种

关联明确化，将这个类和它的协助函数包在同一个名字空间里（8.2节）：

```
namespace Chrono {           // 处理时间的各种功能
    class Date { /* ... */ };
    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

名字空间`Chrono`也可以很自然地包含另外一些相关的类，例如`Time`和`Stopwatch`，以及它们的协助函数。在一个名字空间中只保存一个类通常是过于费神了，也会带来许多不便。

10.3.3 重载的运算符

增加一些具有习惯形式的函数也很有用。例如，下面`operator==`函数的定义使相等运算符`==`也可以用于`Date`：

```
inline bool operator==(Date a, Date b) // 相等
{
    return a.day()==b.day() && a.month()==b.month() && a.year()==b.year();
}
```

其他明显的候选者包括

```
bool operator!=(Date, Date);           // 不等
bool operator<(Date, Date);           // 小于
bool operator>(Date, Date);           // 大于
// ...

Date& operator++(Date& d);             // 将Date增加一天
Date& operator--(Date& d);             // 将Date减少一天

Date& operator+=(Date& d, int n);       // 加n天
Date& operator-=(Date& d, int n);       // 减n天

Date operator+(Date d, int n);          // 加n天
Date operator-(Date d, int n);          // 减n天

ostream& operator<<(ostream&, Date d); // 输出d
istream& operator>>(istream&, Date& d); // 读入到d
```

对于`Date`而言，这些运算符可以只看做是为了方便。但是，对于许多类型——例如复数（11.3节）、向量（3.7.1节）和类似函数的对象（18.4节）——常规运算符的使用已经如此牢固扎根于人们的头脑中，定义它们几乎是绝对必要的。运算符重载问题将在第11章讨论。

10.3.4 具体类型的意义

我将像`Date`这样的简单用户定义类型称做具体类型，是为了使它们能有别于抽象类型（2.5.4节）和类层次结构（12.3节），也是为了强调它们与`int`和`char`等内部类型的相似性。这种类型有时也被称做值类型，而将它们的使用称为面向值的程序设计。这些类型的使用模型以及隐藏在它们的设计背后的“哲学”与那种所谓面向对象程序设计（2.6.2节）的东西之间存在着很大的差异。

具体类型的意图就是使一个单独的、相对较小的事情能够很好地并且高效地完成。为用户提供某种功能，使他们可以修改具体类型的行为，一般而言并不是这里的目标。特别地，通常并不希望具体类型表现出多态性的行为（2.5.5节、12.2.6节）。

如果你不喜欢某个具体类型的一些细节，你可以构造出另一个新的具有所期望行为的东西。如果你想“重用”一个具体类型，你将它用在你自己新类型的实现中，完全就像你使用的是`int`一样。例如，

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    Date_and_time(Date d, Time t);
    Date_and_time(int d, Date::Month m, int y, Time t);
    // ...
};
```

将在第12章里讨论的派生类机制可用于从一个具体类型出发定义新类型，只需描述出所希望的差异。从`vector`定义出`Vec`（3.7.2节）就是这样的一个例子。

如果有一个非常好的编译器，像`Date`这样的具体类型将不会带来隐性的时间或者空间上的额外开销。具体类型的大小在编译时均已知晓，因此这种对象可以在运行栈上分配（也就是说，无须使用自由存储操作）。每个对象的布局在编译时均已知晓，因此，操作的在线化也很容易做到。与此类似，获得与其他语言（例如C或者Fortran）的布局兼容性也不需要做出特别的努力。

定义良好的一组这种类型能够为应用提供一个基础。在一个应用中，如果缺乏一批合适的“小而有效的类型”，并因此去使用过分一般且代价昂贵的类，就可能导致严重的时间与空间上的低效率。换一个角度，缺乏具体类型将导致结构模糊的程序和时间的浪费，因为每个程序员都需要写代码，去直接操作“小而频繁使用的”数据结构。

10.4 对象

对象的建立存在着多种可能方式。有些对象是局部变量，有些是全局变量，有些是类的成员，如此等等。本节将讨论这方面的不同情况、统辖着它们的规则、初始化对象所用的构造函数，以及在它们变得不再可用之前清理它们的析构函数。

10.4.1 析构函数

构造函数完成对象的初始化，换句话说，它建立起一种成员函数将在其中进行操作的环境。有时建立这样的环境涉及到申请某些资源——例如，一个文件、一个锁或者一些存储——这些资源在使用之后必须释放（14.4.7节）。这样，有些类里就需要有一个函数，并要保证它将在每个对象被销毁之前得以调用，就像构造函数能够得到类似保证，在对象创建时必定被调用一样。按照习惯，这个函数被称为析构函数，它通常完成一些清理和释放资源的工作。当一个自动变量离开其作用域时，当一个位于自由存储的对象被删除时，还有在其他类似情况中，析构函数都将被隐式地调用。只有在非常特殊的情形下，用户才需要去显式地调用析构函数（10.4.11节）。

析构函数的最常见用途是为了释放构造函数请求的存储空间。考虑一个以某种类型`Name`为元素的简单的表`Table`。`Table`的构造函数必须为保存其元素分配存储。而到最后，当这种表被删除的时候，我们就必须保证这些存储能收回，以便将来再用。我们做到这一点的方式就是提供一个与构造函数互补的特殊函数：

```
class Name {
    const char* s;
    // ...
};

class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz = s]; } // 构造函数
    ~Table() { delete[] p; }                       // 析构函数
    Name* lookup(const char*);
    bool insert(Name*);
};
```

析构函数的记法形式 `~Table()` 采用求补的符号作为提示，以表明析构函数与构造函数 `Table()` 之间的关系。

一对相互匹配的构造函数/析构函数是在C++里实现大小可以变化的概念的常用机制。标准库容器，例如`map`，就是采用这一技术的某种变形为它们的元素提供存储。所以，下面讨论中所阐释的技术，也就是你每次使用一个标准容器（例如，一个标准`string`）时都实际依靠的东西。这个讨论也适用于那些没有析构函数的类型，可以简单地认为这种类型有着一个什么也不做的析构函数。

10.4.2 默认构造函数

类似地，也可以认为大部分类型都有一个默认构造函数。默认构造函数就是调用时不必提供参数的构造函数。由于带有默认参数15，`Table::Table(size_t)` 也就是默认构造函数。如果用户自己声明了一个默认构造函数，那么就会去使用它；否则，如果有必要，而且用户没有声明其他的构造函数，编译器就会设法去生成一个。编译器生成的默认构造函数将隐式地为类类型^①的成员和它的基类（12.2.2节）调用有关的默认构造函数。例如，

```
struct Tables {
    int i;
    int vi[10];
    Table t1;
    Table vt[10];
};

Tables tt;
```

这里的`tt`将被用一个生成出来的默认构造函数初始化，该构造函数为`tt.t1`以及`tt.vt`的每个成员调用`Table(15)`。在另一方面，它不会去初始化`tt.i`和`tt.vi`，因为它们不是类类型的对象。对于

① class type，即指那些由程序员定义的类型而产生的类型，以便与内部类型和其他用户定义类型相区分。下面常有这一说法，类似的说法如“类对象”等。——译者注

类类型和内部类型区别对待是为了与C的兼容性，也为避免运行时的额外开销。

由于`const`和引用（5.5节、5.4节）必须进行初始化，包含`const`或引用成员类就不能进行默认构造，除非程序员显式地提供了默认构造函数（10.4.6.1节）。例如，

```
struct X {
    const int a;
    const int& r;
};

X x; // 错误：X无默认构造函数
```

默认构造函数也可以显式调用（10.4.10节）。内部类型同样也有默认构造函数（6.2.8节）。

10.4.3 构造和析构

现在考虑建立对象的各种不同方式，以及它们在后来怎样销毁。一个对象可以通过如下方式建立：

- 10.4.4节 一个命名的自动对象，当程序的执行每次遇到它的声明时建立，每次程序离开它所出现的块时销毁。
- 10.4.5节 一个自由存储对象，通过`new`运算符建立，通过`delete`运算符销毁。
- 10.4.6节 一个非静态成员对象，作为另一个类对象的成员，在它作为成员的那个对象建立或销毁时，它也随之被建立或销毁。
- 10.4.7节 一个数组元素，在它作为元素的那个数组被建立或销毁的时候建立或销毁。
- 10.4.8节 一个局部静态对象，在程序执行中第一次遇到它的声明时建立一次，在程序终止时销毁一次。
- 10.4.9节 一个全局对象、名字空间的对象、类的静态对象，它们只在“程序开始时”建立一次，在程序终止时销毁一次。
- 10.4.10节 一个临时对象，作为表达式求值的一部分被建立，在它所出现的那个完整表达式的最后被销毁。
- 10.4.11节 一个在分配操作中由所提供的参数控制，在通过用户提供的函数获得的存储里放置的对象。
- 10.4.12节 一个`union`成员，它不能有构造函数和析构函数。

这个表大致按照重要性排了序。下面各节将解释这些不同的对象创建方式及其使用。

10.4.4 局部变量

对一个局部变量的构造函数将在控制线程每次通过该变量的声明时执行。每次当控制离开该局部变量所在的块时，就会去执行它的析构函数。一组局部变量的析构函数将按构造它们的相反顺序执行。例如，

```
void f(int i)
{
    Table aa;
    Table bb;
    if (i>0) {
        Table cc;
        // ...
    }
```

```

    }
    Table dd;
    // ...
}

```

在这里，每次 $f()$ 被调用时将构造起 aa 、 bb 和 dd （按此顺序）；而每次从 $f()$ 里返回时， dd 、 bb 、 aa 将（按此顺序）被析构。如果对一个调用 $i > 0$ ， cc 将在 bb 之后构造，并在 dd 的构造之前析构。

10.4.4.1 对象的复制

如果 $t1$ 和 $t2$ 都是类 $Table$ 的对象， $t1 = t2$ 的默认含义就是将 $t1$ 按成员逐个复制到 $t2$ （10.2.5节）。对于赋值的这种解释方式，在应用到具有指针成员的类的对象时，就可能产生一种出人意料的（通常也是人们所不希望）的作用。对于包含了由构造函数/析构函数管理的资源的对象而言，按成员复制的语义通常是不正确的。例如，

```

void h()
{
    Table t1;
    Table t2 = t1;  // 复制初始化：麻烦
    Table t3;

    t3 = t2;        // 复制赋值：麻烦
}

```

在这里， $Table$ 的默认构造函数为 $t1$ 和 $t3$ 各调用了一次，一共两次。然而 $Table$ 的析构函数则被调用了3次：对 $t1$ 、 $t2$ 和 $t3$ 各一次！由于赋值的默认解释是按成员赋值，所以，在 $h()$ 结束时， $t1$ 、 $t2$ 和 $t3$ 中将各包含一个指针，它们都指向建立 $t1$ 时从自由存储中分配的那个名字数组。在建立 $t3$ 时所分配的数组的指针并没有保留下来，因为它被赋值 $t3 = t2$ 覆盖掉了。这样，如果没有自动废料收集（10.4.5节），对这个程序而言，该数组的存储就将永远丢掉了。而在另一方面，为 $t1$ 的创建而分配的数组因为同时出现在 $t1$ 、 $t2$ 和 $t3$ 里，将被删除3次。这种情况所导致的结果是无定义的，很可能是灾难性的。

这类反常情况可以避免，方式就是将 $Table$ 复制的意义定义清楚：

```

class Table {
    // ...
    Table(const Table&);           // 复制构造函数
    Table& operator=(const Table&); // 复制赋值
};

```

程序员可以为这些复制操作定义自己认为最合适的任何意义，但对于这类容器，传统上就是复制那些被包含的元素（或至少是让用户以为做了复制，参见11.12节）。例如，

```

Table::Table(const Table& t)      // 复制构造函数
{
    p = new Name[sz=t.sz];
    for (int i = 0; i < sz; i++) p[i] = t.p[i];
}

Table& Table::operator=(const Table& t) // 赋值
{
    if (this != &t) {             // 当心自赋值：t = t
        delete[] p;
        p = new Name[sz=t.sz];
        for (int i = 0; i < sz; i++) p[i] = t.p[i];
    }
}

```

```

    }
    return *this;
}

```

情况几乎总是如此，赋值构造函数与复制赋值通常都很不一样。究其根本原因，复制构造函数是去完成对未初始化的存储区的初始化，而复制赋值运算符则必须正确处理一个结构良好的对象。

在某些情况下，可以对赋值做一些优化，但赋值运算符的一般性策略非常简单：防止自赋值，删除那些老元素，初始化，复制那些新元素。通常每个非静态的成员都必须复制（10.4.6.3节）。可以通过异常去报告复制中出错的情况（附录E）。

10.4.5 自由存储

在自由存储中建立对象时，有关的构造函数由`new`调用，这个对象将一直存在直到将`delete`函数作用于指向它的指针。考虑

```

int main()
{
    Table* p = new Table;
    Table* q = new Table;

    delete p;
    delete p; // 可能导致运行时错误
}

```

构造函数`Table::Table()`将被调用两次，析构函数`Table::~~Table()`也一样。不幸的是，在这个例子中的`new`和`delete`不匹配，致使`p`所指的对象被删除了两次，而`q`所指的对象却根本没删除。如果只考虑语言的话，忘记删除某个对象一般不算是错误，这样做只会浪费存储。当然，如果一个程序原本就是想长时间地运行，这种存储流失就会成为很严重的问题，而且是极难发现的错误。存在着一些能够检查这种流失的工具。删除`p`两次则是很严重错误，其行为没有定义，很可能是灾难性的。

有的C++实现能够自动地重复使用由无法到达的对象所占据的那些存储（带废料收集的实现），但这种方式并非标准。即使存在一个正在运行的废料收集器，`delete`也会去调用析构函数（如果存在的话），因此，两次删除一个对象仍然是灾难性的。在许多情况下，这里只有一点小小的不方便。特别是，如果知道存在有废料收集器，那些只做存储管理的析构函数就都可以去掉了，这种简化所付出的代价是可移植性，对某些程序可能增加运行时间，并有可能丧失对于其运行时间性质的可预见性（C.9.1节）^①。

将`delete`应用于某个对象之后，再以任何方式访问该对象都是错误的。不幸的是，实现将无法去检查这种错误。

用户还可以描述`new`如何分配存储，`delete`如何回收（6.2.6.2节和15.6节）。也可以特别说明分配、初始化（构造）和异常处理之间的交互情况（14.4.5节和19.4.5节）。有关在自由空间分配数组的问题将在10.4.7节讨论。

① 废料收集可能在任何时刻发生，许多废料收集系统工作时要求将其他工作（如系统的正常处理）停下来，直到它的工作完成，这将造成系统正常处理的中断。这样，对于系统中任何操作，只要它在执行中申请存储，其间就可能出现废料收集，这也将使人无法预计该操作需要多少时间才能完成。——译者注

10.4.6 类对象作为成员

现在考虑一个保存有关小型组织的信息的类

```
class Club {
    string name;
    Table members;
    Table officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};
```

Club的构造函数以组织的名称及其创建时间为参数，提供给成员的构造函数的参数在一个成员初始式到表中描述，写在这个容器类的构造函数定义中。例如，

```
Club::Club(const string& n, Date fd)
    : name(n), members(), officers(), founded(fd)
{
    // ...
}
```

成员初始式列表由一个冒号开头，用逗号分隔。

成员的构造函数将在容器类本身的构造函数的体执行之前首先被执行。这些构造函数按照成员在类中声明的顺序执行，而不是按这些成员在初始式表中出现的顺序。为了避免混乱，最好还是按照各成员的声明顺序描述这些初始式。在类本身的析构函数体执行之后，各成员的析构函数将按照与构造相反的顺序被逐个调用。

如果某个成员的构造函数不需要参数，在初始式表里就无须提到这个成员，所以

```
Club::Club(const string& n, Date fd)
    : name(n), founded(fd)
{
    // ...
}
```

与前面的描述等价。在两种情况中，**Table::Table**都将用默认值15调用**Club::officers**的构造函数。

如果某个类对象中包含着一些类对象，当该对象被销毁时，它自己的析构函数（如果有的话）将首先执行，而后将按照与声明相反的顺序执行各个成员的析构函数。构造函数自下而上地（成员在先）为成员函数装配起执行环境，而析构函数则以自上而下（成员在后）的方式拆除它。

10.4.6.1 成员初始化的必要性

对于那些初始化与赋值不同的情况——即对于那些没有默认构造函数的类的成员对象，对于那些**const**成员和引用成员而言，对成员的初始式都是必不可少的。例如，

```
class X {
    const int i;
    Club c;
    Club& pc;
    // ...
    X(int ii, const string& n, Date d, Club& c) : i(ii), c(n,d), pc(c) {}
};
```

不存在对这些成员做初始化的其他方式，而且，不对这种成员做初始化就是错误。当然，对于大部分类型而言，程序员可以选择使用初始式或者使用赋值。对于这类情况，我还是喜欢采用初始式的语法形式，因为这使已做了初始化的情况更加明显。此外，使用初始化语法形式还可能带来效率上的优势。例如，

```
class Person {
    string name;
    string address;
    // ...
    Person(const Person&);
    Person(const string& n, const string& a);
};

Person::Person(const string& n, const string& a)
    : name(n)
{
    address = a;
}
```

这里的`name`用`n`的一个副本进行初始化。而在另一方面，对`address`将先用一个空串初始化，而后又用`a`的副本赋值。

10.4.6.2 成员常量

对那些静态整型成员，可以给它的成员声明加上一个常量表达式作为初始式。例如，

```
class Curious {
public:
    static const int c1 = 7;           // ok; 但要记得去定义
    static int c2 = 11;               // 错误: 非const
    const int c3 = 13;                // 错误: 非static
    static const int c4 = f(17);       // 错误: 在类里的初始表达式不是常量
    static const float c5 = 7.0;       // 错误: 在类里初始化的不是整型
    // ...
};
```

当（且仅当）你用到某个被初始化的成员，而且需要将它作为对象存入存储器时，这个成员就必须在某处有（惟一的）定义。初始式不必重复写：

```
const int Curious::c1;                // 必须，但这里不必重复初始式
const int* p = &Curious::c1;        // ok: Curious::c1已经有定义
```

换一种方式，你也可以在类声明中用枚举符（4.8节、14.4.6节、15.3节）作为符号常量。例如，

```
class X {
    enum { c1 = 7, c2 = 11, c3 = 13, c4 = 17 };
    // ...
};
```

不要试图在类里用这种方式去初始化变量和浮点数等。

10.4.6.3 成员的复制

默认的复制构造函数或复制赋值（10.4.4.1节）做的就是对类中所有成员的复制。如果这种复制无法进行，试图复制这种类的成员就是一个错误。例如，

```
class Unique_handle {
```

```

private:      // 复制操作作为私用, 禁止复制 (11.2.2节)
    Unique_handle(const Unique_handle&);
    Unique_handle& operator=(const Unique_handle&);
public:
    // ...
};

struct Y {
    // ...
    Unique_handle a;    // 要求显式的初始化
};

Y y1;
Y y2 = y1;    // 错误: 无法复制Y::a

```

此外, 如果存在着某个非静态成员是一个引用、*const*或者属于没有复制赋值的用户定义类型, 那么也无法生成默认的复制赋值。

请注意, 默认的复制构造函数会导致原对象和复制出来的对象的引用成员引用着同一个对象, 如果被引用的对象后来被删除, 那就会造成问题。

在写复制构造函数时, 我们必须小心地去复制每一个需要复制的元素。如果不写, 那么就按默认方式做元素的初始化, 而这常常不是复制构造函数中所需的方式。例如,

```

Person::Person(const Person& a) : name(a.name) { }    // 当心!

```

在这里我忘了复制*address*, 因此就使*address*被按照默认方式用空串初始化。在为一个类增加新成员时, 总应该去检查, 看是不是有用户定义的构造函数需要更新, 以便能完成对新成员的初始化和复制。

10.4.7 数组

如果在构造某个类的对象时可以不提供显式的初始式, 那么就可以定义这个类的数组。例如,

```

Table tbl[10];

```

这将建立起一个10个*Table*的数组, 并用默认参数15调用*Table::Table()*, 对每个*Table*进行初始化。

除了使用初始式列表 (5.2.1节、18.6.7节) 之外, 没有其他方式能为数组声明中的构造函数提供显式参数。如果你绝对需要将某个数组的各成员初始化为不同的值, 那么你可以写一个默认构造函数, 让它去产生所想要的值。下面是一个例子

```

class Ibuffer {
    string buf;
public:
    Ibuffer() { cin>>buf; }
    // ...
};

void f()
{
    Ibuffer words[100]; // 由cin初始化words的各元素
    // ...
}

```

最好还是避免这种过于精妙的东西。

当一个数组被销毁时，它将对数组中构造起的各个元素调用析构函数。对于那些不是通过`new`分配的数组，这件事情将隐式地完成。像C++一样，C++不区分指向单个对象的指针和指向某数组起始元素的指针（5.3节）。由于这种情况，程序员必须明确说明要删除的是数组还是单个对象。例如，

```
void f(int sz)
{
    Table* t1 = new Table;
    Table* t2 = new Table[sz];
    Table* t3 = new Table;
    Table* t4 = new Table[sz];

    delete t1;      // 正确
    delete[] t2;    // 正确
    delete[] t3;    // 错：麻烦
    delete t4;      // 错：麻烦
}
```

数组和单个对象的确切分配方式是由具体实现确定的。因此，不同的实现对于错误使用`delete`和`delete[]`操作的反应也可能不同。对于那些简单而无趣的情况，例如上面的例子，某个编译器有可能发现问题。但一般来说，这会在运行时造成某种极其险恶的后果。

针对数组的特殊析构运算符，`delete[]`，在逻辑上并不必要。当然，那样做的条件是自由存储的实现必须在每个对象里保存足够的信息，告知它是单个对象还是数组。如果真的采用这一做法，用户可以减轻一些负担，但这种义务也可能给某些C++实现带来时间和空间上非常明显的额外开销。

如常，如果你觉得C风格的数组过于麻烦，请使用诸如`vector`（3.7.1节、16.3节）这样的类代替之。例如，

```
void g()
{
    vector<Table>* p1 = new vector<Table>(10);
    Table* p2 = new Table;

    delete p1;
    delete p2;
}
```

10.4.8 局部静态存储

对于局部静态对象（7.1.2节）而言，构造函数是在控制线程第一次通过该对象的定义时调用。考虑下面程序段

```
void f(int i)
{
    static Table tbl;
    // ...
    if (i) {
        static Table tbl2;
        // ...
    }
}
```

```
int main()
{
    f(0);
    f(1);
    f(2);
    // ...
}
```

在这里，在`f()`第一次调用时将为`tbl`调用一次构造函数。因为`tbl`被声明为`static`，在`f()`返回时它并不销毁，而且在`f()`再次调用时也不会第二次去构造它。由于包含`tbl2`的声明的块在调用`f(0)`中不执行，因此，在`f(1)`调用之前`tbl2`不会被构造，在第二次执行它所在的块时也不会再次去构造它。

在程序结束时，局部静态对象的析构函数将按照它们被构造的相反顺序逐一调用（9.4.1.1节），没有规定确切的时间。

10.4.9 非局部存储

在所有函数之外定义的变量（即全局变量、名字空间的变量，以及各个类的`static`变量；C.9节）在`main()`被激活之前完成初始化（构造），对于已经构造起的所有这些变量，其析构函数将在退出`main()`之后调用。动态连接机制使这里的情景稍微复杂了一点，这种情况下的初始化将延迟至代码连接到程序上的时候进行^①。

在一个编译单位里，对非局部变量的构造将按照它们的定义顺序进行。考虑

```
class X {
    // ...
    static Table memtbl;
};

Table tbl;

Table X::memtbl;

namespace Z {
    Table tbl2;
}
```

构造的顺序是先`tbl`，而后是`X::memtbl`，再往后是`Z::tbl2`。注意，如在`X`里的`memtbl`声明一类的声明（与定义不同）并不影响构造的顺序。析构函数将按照与构造相反的顺序逐个调用，先`Z::tbl2`，而后是`X::memtbl`，最后是`tbl`。

对于不同编译单位里的非局部变量，其构造顺序就没有保证了，完全依赖于具体的实现。例如，

```
// file1.c:
    Table tbl1;

// file2.c:
    Table tbl2;
```

① 在支持动态连接方式的环境里，可以让一些代码段在程序运行过程中与程序的基本部分连接。如果这些代码段包含非局部变量，这些变量的初始化就将在代码段连接时进行。这个规定不会影响构成程序的那些静态连接的代码段里的非局部变量，它们仍在`main()`执行之前完成初始化。——译者注

到底`tbl1`是在`tbl2`之前还是之后构造，这完全由实现确定。一个特定实现甚至不保证总采用某种固定的顺序。动态连接、编译过程中的小变动等都可能改变这种顺序。析构的顺序也与此类似，也依赖于实现。

确实有这样的情况，在你设计一个库时，有时为了需要，或者就是为了方便，创建了一个专门为了初始化和清理的带有构造函数和析构函数的类型。这种类型应该只使用一次：分配一个静态对象以调用构造函数和析构函数。看下面例子：

```
class Zlib_init {
    Zlib_init();    // 使Zlib能够使用
    ~Zlib_init();   // 对Zlib做最后的清理
};

class Zlib {
    static Zlib_init x;
    // ...
};
```

不幸的是，在一个由若干分别编译单位组成的程序里，无法保证这种对象一定能在它的第一次使用之前初始化，在其最后的使用之后销毁。特定的C++ 实现有可能提供某类保证，但大部分实现都不是这样的。程序员可以通过下面的实现策略来保证正确的初始化，即采用一种通常用于局部静态对象的技术：第一次开关。例如，

```
class Zlib {
    static bool initialized;
    static void initialize() { /* 初始化 */ initialized = true; }
public:
    // 无构造函数

    void f()
    {
        if (initialized == false) initialize();
        // ...
    }
    // ...
};
```

如果有许多函数都需要检测第一次开关，这一做法也可能令人厌烦，不过还是可以控制的。这种技术依赖于一个事实：无构造函数的静态分配对象将自动地初始化为0。真正困难的情况是那些第一次操作的时间要求极其严格的情况，对于它们检测和可能的初始化所造成的额外开销都将成为严重问题。对于那些情况，就需要进一步的技巧了（21.5.2节）。

对于简单对象的另一种解决方法是将它表示为一个函数（9.4.1节）

```
int& obj() { static int x = 0; return x; } // 第一次使用时初始化
```

第一次开关不可能处理能够想到的所有情况。例如，完全可能建立起在构造中相互引用的对象。当然，这种例子最好还是避免。如果确实需要这种对象，那么就应该分步骤地小心地将它们建立起来。另外，并没有与此类似的最后时间开关结构。参见9.4.1.1节和21.5.2节。

10.4.10 临时对象

临时对象最经常是作为算术表达式的结果出现的。例如，在求值 $x * y + z$ 过程中的某一点，

部分结果 $x * y$ 必须存在于某个地方。除了与性能有关的问题之外（11.6节），临时对象很少成为程序员关心的问题。但这种情况确实也可能出现（11.6节、22.4.7节）。

除非一个临时对象被约束到某个引用，或者被用于做命名对象的初始化，否则它将总在建立它的那个完整表达式结束时销毁。所谓完整表达式就是那种不是其他表达式的子表达式的表达式。

标准`string`类里有一个成员函数`c_str()`，它返回一个C风格的以0结尾的字符数组（3.5.1节、20.4.1节）。此外，`+`在这里定义为串拼接。这些都是对`string`很有用的功能。但是，它们的组合使用却可能导致很隐晦的问题。例如，

```
void f(string& s1, string& s2, string& s3)
{
    const char* cs = (s1+s2).c_str();
    cout << cs;

    if (strlen(cs=(s2+s3).c_str()) < 8 && cs[0] == 'a') {
        // 在这里用cs
    }
}
```

你的第一个反应可能是“千万不要这样写”，我也同意。然而，这种代码还是被一些人写了出来，因此值得去理解对于它的解释。

为了保存`s1 + s2`将会产生一个临时对象，随之从这个对象里提取出一个指向C风格字符串的指针。再往后——在表达式结束时——这个临时对象将被删除。那么由`c_str()`返回的那个C风格字符串在什么地方呢？或许它是作为保存`s1 + s2`的临时对象的一部分，而在这个临时对象被销毁后，根本无法保证这个存储还存在。这样`cs`就会指向已经释放掉的存储。输出操作`cout << cs`可能像所希望的那样工作，但那只能是侥幸。编译器有可能检查出这个问题的许多变形，并给出警告。

与if语句有关的例子琢磨起来就更困难一点。条件将如预期的那样工作，因为建立起保存`s2 + s3`的临时对象的完整表达式就是整个的条件。但是，在进入受控语句之前，这个临时变量已经被销毁了，在那里，任何对`cs`的使用都不能保证工作。

请注意，在这些情况和许多类似的情况中，与临时变量有关的问题都出在以低级方式使用高级的数据类型。一种更清晰的程序设计风格不仅能产生更易于理解的程序段，而且也完全避免临时对象。例如，

```
void f(string& s1, string& s2, string& s3)
{
    cout << s1+s2;
    string s = s2+s3;

    if (s.length() < 8 && s[0] == 'a') {
        // 在这里使用s
    }
}
```

可以用临时对象作为`const`引用或者命名对象的初始式。例如，

```
void g(const string&, const string&);

void h(string& s1, string& s2)
{
```

```

    const string& s = s1+s2;
    string ss = s1+s2;

    g(s,ss); // 我们可以在这里用s和ss
}

```

这样做都没有问题。这种临时对象将在“它的”引用或者命名变量离开作用域时销毁。请记住，返回索引到局部变量的引用也是错误的（7.3节），临时对象也不能被约束到非`const`引用（5.5节）。

也可以通过显式地调用构造函数的方式建立临时对象，例如，

```

void f(Shape& s, int x, int y)
{
    s.move(Point(x,y)); // 构造一个Point并传递给Shape::move()
    // ...
}

```

这种临时对象的销毁方式与隐式生成的临时对象完全一样^①。

10.4.11 对象的放置

按照默认方式，运算符`new`将在自由存储中创建对象。如果我们需要在其他地方分配对象，那又该怎么办呢？考虑一个简单类：

```

class X {
public:
    X(int);
    // ...
};

```

通过提供一个带额外参数的分配函数，而后在使用`new`时提供对应参数的方式，我们可以将对象放到任何地方：

```

void* operator new(size_t, void* p) { return p; } // 显式放置运算符

void* buf = reinterpret_cast<void*>(0xF00F); // 重要地址
X* p2 = new(buf) X; // 在“buf”构造X时调用：operator new(sizeof(X), buf)

```

正是由于这种使用方式，为`operator new`提供额外参数的`new (buf) X`的这种语法形式被称做放置语法。注意，每个`new`总以对象的大小作为其第一个参数，而被分配对象的大小是隐式提供的（15.6节）。由`new`运算符所使用的`operator new()`按普通的参数匹配规则选择（7.4节）；每个`operator new()`的第一个参数都是`size_t`。

“放置式”`operator new()`是这类分配器的最简单情况，在标准头文件`<new>`里定义。

`reinterpret_cast`是最粗鲁，或许也是最肮脏的类型转换运算符（6.2.7节）。在大部分情况下，它就是简单地产生一个值，其二进制位模式与原参数完全一样，并且具有所需要的类型。这样，该运算符可以用于那些本质上就是依赖于实现的、危险的，而偶然又是绝对必需的将整数转换到指针或反过来的活动。

放置式的`new`结构也可以用于从某个特定的场地中分配存储

① 请注意，在这里，建立临时变量的“完整”表达式是“`s.move(Point(x,y))`”。按规则，所建立的临时变量将在这个完整的表达式结束时销毁，是在该函数调用返回之时。——译者注


```

class Arena {
public:
    virtual void* alloc(size_t) =0;
    virtual void free(void*) =0;
    // ...
};

void* operator new(size_t sz, Arena* a)
{
    return a->alloc(sz);
}

```

现在就可以在不同Arena（场地）里分配任意类型的对象了。例如，

```

extern Arena* Persistent;
extern Arena* Shared;

void g(int i)
{
    X* p = new(Persistent) X(i);    // 在某持续性(Persistent)存储分配X
    X* q = new(Shared) X(i);       // 在共享的(Shared)存储分配X
    // ...
}

```

将对象放置在某个分配场地的活动不（直接）由标准的自由存储管理器控制，这也意味着在销毁对象时需要当心。完成此事的基本机制就是显式地调用析构函数：

```

void destroy(X* p, Arena* a)
{
    p->~X();    // 调用析构函数
    a->free(p); // 释放存储
}

```

请注意，这里对析构函数的显式调用，就像使用专用的全局存储分配器一样，只要有可能就应该避免这种东西。但是，偶尔也有一些情况下必须这样做。例如，如果不通过显式的析构函数调用，我们将很难按照标准库vector的方式（3.7.1节、16.3.8节）去实现有效的通用容器类。但无论如何，作为新手，在显式调用析构函数之前还是应该思量再三，并应该向更有经验的同事做一些咨询。

关于放置式的new与异常处理之间的相互作用，请看14.4.4节的解释。

没有针对数组的特殊放置语法，也没有这种必要，因为任意类型都能用放置式的new分配。但是可以为数组定义一个特殊的operator delete()（19.4.5节）。

10.4.12 联合

命名联合的定义方式同struct，其中的各个成员将具有同样的地址（C.8.2节）。联合可以有成员函数，但却不能有静态成员。

一般来说，编译器无法知道被使用的是联合的哪个成员；也就是说，无法知道存储在联合中的对象的类型。因此，联合就不能包含带构造函数或析构函数的成员，因为无法保护其中的对象以防止破坏，也不可能保证在联合离开作用域时能调用正确的析构函数。

最好是仅仅将联合用于底层代码，或者作为某个类的实现中的一部分，由这个类去维护有关在联合中存储着什么的信息（10.6[20]）。

10.5 忠告

- [1] 用类表示概念；10.1节。
- [2] 只将**public**数据（*struct*）用在它实际上仅仅是数据，而且对于这些数据成员并不存在不变式的地方；10.2.8节。
- [3] 一个具体类型属于最简单的类。如果适用的话，就应该尽可能使用具体类型，而不要采用更复杂的类，也不要简单的数据结构；10.3节。
- [4] 只将那些需要直接访问类的表示的函数作为成员函数；10.3.2节。
- [5] 采用名字空间，使类与其协助函数之间的关系更明确；10.3.2节。
- [6] 将那些不修改对象值的成员函数做成**const**成员函数；10.2.6节。
- [7] 将那些需要访问类的表示，但无须针对特定对象调用的成员函数做成**static**成员函数；10.2.4节。
- [8] 通过构造函数建立起类的不变式；10.3.1节。
- [9] 如果构造函数申请某种资源，析构函数就应该释放这一资源；10.4.1节。
- [10] 如果在一个类里有指针成员，它就需要有复制操作（包括复制构造函数和复制赋值）；10.4.4.1节。
- [11] 如果在一个类里有引用成员，它就可能需要有复制操作（复制构造函数和复制赋值）；10.4.6.3节。
- [12] 如果一个类需要复制操作或析构函数，它多半还需要有构造函数、析构函数、复制赋值和复制构造函数；10.4.4.1节。
- [13] 在复制赋值里需要检查自我赋值；10.4.4.1节。
- [14] 在写复制构造函数时，请小心地复制每个需要复制的元素（当心默认的初始式）；10.4.4.1节。
- [15] 在向某个类中添加新成员时，一定要仔细检查，看是否存在需要更新的用户定义构造函数，以使它能够初始化新成员；10.4.6.3节。
- [16] 在类声明中需要定义整型常量时，请使用枚举；10.4.6.2节。
- [17] 在构造全局的和名字空间的对象时，应避免顺序依赖性；10.4.9节。
- [18] 用第一次开关去缓和顺序依赖性问题；10.4.9节。
- [19] 请记住，临时对象将在建立它们的那个完整表达式结束时销毁；10.4.10节。

10.6 练习

- 1. (*1) 找出10.2.2节**Date::add_year()**里的一个错误。在10.2.7节的那个版本里找出另外两个错误。
- 2. (*2.5) 完成**Date**并进行测试。用“1970年1月1日之后的天数”的表示方式重新实现它。
- 3. (*2) 从商业应用中找出一个**Date**，对它所提供的功能做出批判和评论。如果可能的话，再去与一个实际用户讨论这个**Date**。
- 4. (*1) 你如何从名字空间**Chrono**（10.3.2节）中的**Date**类里访问**set_default**？请给出至少三种不同方式。
- 5. (*2) 定义类**Histogram**（直方图），它保存由其构造函数的参数所指定的区间里的计数值。

提供函数打印*Histogram*里的计数值。设法处理越界的情况。

6. (*2) 定义一些类, 它们能提供符合某种分布的随机数值 (例如, 均匀分布和指数分布), 每个类都有一个构造函数刻画分布的参数, 另有一个函数*draw*返回下一个随机值。
7. (*2.5) 完成保存 (名字, 值) 对偶的*Table*。然后修改6.1节的桌面计算器, 让它使用*Table*而不是*map*。从各个方面比较这两个版本。
8. (*2) 将7.10[7] 的*Tnode*重写为一个有构造函数和析构函数等的类。定义一个带有构造函数和析构函数等的*Tnode*的树类。
9. (*3) 定义、实现并测试一个整数集合的类*Intset*, 提供并、交和对称差等操作。
10. (*1.5) 将类*Intset*修改为一个结点*Node*的集合, 其中*Node*是你定义的一种*struct*。
11. (*1) 定义一个类去做简单算术表达式的分析、存储、求值和打印, 表达式由整常数和运算符 +、-、*、/ 构成。类的公用界面看起来具有下面样子:

```
class Expr {
    // ...
public:
    Expr(const char*);
    int eval();
    void print();
};
```

构造函数*Expr::Expr()* 的字符串参数就是有关的表达式。函数*Expr::eval()* 返回表达式的值, *Expr::print()* 在*cout*打印出表达式的某种表示。程序可能具有下面的样子:

```
Expr x("123/4+123*4-3");
cout << "x = " << x.eval() << "\n";
x.print();
```

定义*Expr*类两次, 一次用结点的链接表作为表示, 另一次用字符串作为表示。试验以不同方式打印表达式: 完全加括号的形式、后缀形式、汇编代码等。

12. (*2) 定义类*Char_queue*, 使其公用界面并不依赖于表示。将*Char_queue*实现为 a) 一个链接表; b) 一个向量。不必考虑并发问题。
13. (*3) 为某个语言设计一个符号表类和一个符号表项类。设法去看一看某种语言的一个编译器, 看实际的符号表是什么样的。
14. (*2) 修改10.6[11] 的表达式类, 处理变量和赋值运算符。使用10.6[13] 类的符号表。
15. (*1) 给出程序:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

修改它, 使之产生输出:

```
Initialize
Hello, world!
Clean up
```

但不要对*main()*做任何修改。

16. 定义一个**Calculator**类，采用6.1节的计算器函数作为其主要实现。建立起一些**Calculator**，使它们分别从**cin**、命令行参数或程序里的字符串输入。让输出也可以指向各种目标，就像能够从各种各样的来源输入一样。
17. (*2) 定义两个类，它们各有一个**static**成员，而且每个**static**成员的构造涉及到对另一个的引用。在实际代码里的什么地方可能涉及这种结构？怎样修改这些类以清除构造函数中的顺序依赖性？
18. (*2.5) 比较类**Date**（10.3节）和你对5.9[13]和7.10[19]的解。讨论这两种解决方案在维护中发现错误和其他类似问题方面的差异。
19. (*3) 写一个函数，给它一个**istream**和一个**vector<string>**，它生成一个**map<string, vector<int>>**，其中保存的是各个**string**以及这些**string**所出现的行号。运行这个程序，处理不少于1 000行的正文文件和不少于10个单词。
20. (*2) 从C.8.2节取来**Entry**类，修改它，使每个联合成员总能按照正确的类型被使用。

第11章 运算符重载

当我用一个词时，它的意思就
恰如我的需要——既不多也不少。
——不倒翁

记法——运算符函数——二元和一元运算符——运算符的预定义意义——运算符的用户定义意义——运算符和名字空间——一个复数类型——成员运算符和非成员运算符——混合模式算术——初始化——复制——转换——文字量——协助函数——转换运算符——歧义性解析——友元——成员和友元——大型对象——赋值和初始化——下标——函数调用——间接——增量和减量——一个字符串类——忠告——练习

11.1 引言

每个技术领域——以及大部分非技术领域——都发展出了一套习惯性的简化记述形式，以便使涉及到常用概念的说明和讨论更加方便。例如，由于长期相熟

$x+y*z$

对于我们而言比

multiply y by z and add the result to x

更清楚。这种简洁的常用运算符的重要性，怎么评价都不过分。

与大部分语言一样，C++ 为其内部类型提供了一组运算符。然而，我们希望能够很方便地应用运算符的大部分概念都不是C++ 里的内部类型，这些概念必须用用户定义类型表示。例如，你在C++ 里可能需要复数算术、矩阵代数、逻辑信号、字符串等，你需要用类去表示这些概念。为这些类定义各种运算符，使程序员可以为操作各种对象提供某些记法形式，可能地采用简单的函数记法形式更方便，也更符合人的习惯。例如，

```
class complex {           // 极简单的复数
    double re, im;
public:
    complex(double r, double i) : re(r), im(i) { }
    complex operator+(complex);
    complex operator*(complex);
};
```

这里定义的是复数概念的一个简单实现。一个`complex`被表示为一对双精度浮点数，由运算符`+`和`*`操作。程序员通过定义`complex::operator+()`和`complex::operator*()`，分别为`+`和`*`提供意义。例如，若`b`和`c`都是`complex`类型的，`b + c`的意思就是`b.operator+(c)`。我们现在已经接近`complex`表达式的习惯解释

```

void f()
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;

    a = b+c;
    b = b+c*a;
    c = a*b+complex(1, 2);
}

```

在这里，普通的优先级规则仍然有效。所以，第二个语句的意思就是 $b = b + (c * a)$ 而不是 $b = (b + c) * a$ 。

运算符重载的最明显应用大部分都是针对具体类型的（10.3节）。然而，用户定义运算符的效用并不仅限于各种具体类型。例如，通用的抽象界面的设计中也常常牵涉到一些运算符的使用，如 \rightarrow 、 $[]$ 和 $()$ 等。

11.2 运算符函数

可以为下面各个运算符（6.2节）声明对应的函数定义：

+	-	*	/	%	^	&
	~	!	=	<	>	+=
--	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

下面几个运算符不能由用户定义：

- ::（作用域解析；4.9.4节、10.2.4节）。
- .
- .*（通过到成员的指针做成员选择；15.5节）。

它们都以名字（而不是值）作为其第二个参数，而且提供的是引用成员的最基本语义。允许对它们重载将带来难以琢磨的问题 [Stroustrup, 1994]。三元条件运算符 $?:$ （6.3.2节）也不能重载，还有命名运算符 `sizeof`（4.6节）和 `typeid`（15.4.4节）也是如此。

这里不允许定义新的运算符单词，在运算符集合不够时，你可以采用函数调用的记法形式，例如，用 `pow()` 而不是 $**$ 。这种限制可能看起来太严格，但是采用更灵活的规则很容易引起歧义。例如，定义运算符 $**$ 表示求幂看起来很显然，粗粗一看也非常简单。但请再想一想， $**$ 应该向左约束（如Fortran那样）还是应该向右约束（如Algol那样）？表达式 $a ** p$ 应该解释为 $a * (*p)$ 还是 $(a) ** (p)$ 呢？

运算符函数的名字是由关键字 `operator` 后跟对应的运算符构成的；例如 `operator<<`。运算符函数的定义和使用都可以像其他函数一样。使用运算符不过是显式调用运算符函数的一种简写形式。例如，

```

void f(complex a, complex b)
{
    complex c = a + b;           // 简写
    complex d = a.operator+(b); // 显式调用
}

```

有了前面的`complex`定义, 这两个初始式意义相同。

11.2.1 二元和一元运算符

二元运算符可以定义为取一个参数的非静态成员函数, 也可以定义为取两个参数的非成员函数。对于任何二元运算符`@`, `aa@bb`可以解释为`aa.operator@(bb)`、或者解释为`operator@(aa, bb)`。如果两者都有定义, 那么就按照重载解析(7.4节)来确定究竟应该用哪个定义(如果确有一个该用的话)。例如,

```
class X {
public:
    void operator+(int);
    X(int);
};

void operator+(X, X);
void operator+(X, double);

void f(X a)
{
    a+1;      // a.operator+(1)
    1+a;      // ::operator+(X(1), a)
    a+1.0;    // ::operator+(a, 1.0)
}
```

对于一元运算符, 无论它是前缀的还是后缀的, 都可以定义为无参数的非静态成员函数, 或者定义为取一个参数的非成员函数。对任何前缀一元运算符`@`, `@aa`可以解释为`aa.operator@()`或者`operator@(aa)`。如果两者都有定义, 那么就按照重载解析(7.4节)确定应该使用哪个定义(如果确有一个该用的话)。对任何后缀一元运算符`@`, `aa@`可以解释为`aa.operator@(int)`或者`operator@(aa, int)`(这将在11.11节做进一步解释)。如果这两者都有定义, 那就按照重载解析(7.4节)去确定应该使用哪个定义(如果确有一个该用的话)。运算符只能按照语法定义的形式(A.5节)进行声明。例如, 用户不能去定义一元的`%`或者三元的`+`。考虑

```
class X {
    // 成员函数(有隐式的this指针):

    X* operator&();      // 前缀一元&(取地址)
    X operator&(X);      // 二元&(与)
    X operator++(int);   // 后缀增量(参见11.11节)
    X operator&(X, X);   // 错误: 三元
    X operator/();       // 错误: 一元
};

// 非成员函数:

X operator-(X);          // 前缀一元减
X operator-(X, X);       // 二元减
X operator--(X&, int);   // 后缀减量
X operator-();           // 错误: 无操作数
X operator-(X, X, X);    // 错误: 三元
X operator%(X);          // 错误: 一元%
```

运算符`[]`在11.8节描述, 运算符`()`在11.9节描述, 运算符`->`在11.10节描述, 运算符`++`和

-- 在11.11节描述, 存储分配和释放运算符在6.2.6.2节、10.4.11节和15.6节描述。

11.2.2 运算符的预定义意义

对于用户定义运算符只做了不多的几个假定。特别是`operator=`、`operator[]`、`operator()`和`operator->`只能作为非静态的成员函数, 这就保证了它们的第一个运算对象一定是一个左值(4.9.6节)。

有些内部运算符在意义上等价于针对同样参数的另外一些运算符的组合。例如, 如果`a`是`int`, `++a`的意思就是`a += 1`, 这转而又表示`a = a + 1`。对于用户定义运算符而言就没有这类关系, 除非用户正好将它们定义成这样。例如, 编译器也不会从`Z::operator+()`和`Z::operator=()`的定义去生成出一个`Z::operator+=()`的定义。

由于历史的偶然性, 运算符`=`(赋值)、`&`(取地址)和`,`(序列; 6.2.2节)在应用于类对象时已经有了预先定义的意义。通过将它们定义为私用, 就可以使普通用户无法访问这些预定义的意义:

```
class X {
private:
    void operator=(const X&);
    void operator&();
    void operator,(const X&);
    // ...
};
void f(X a, X b)
{
    a = b;      // 错误: operator= 为私用
    &a;         // 错误: operator& 为私用
    a, b;       // 错误: operator, 为私用
}
```

另一方面, 通过适当定义也可以给它们以新的意义。

11.2.3 运算符和用户定义类型

一个运算符函数必须或者是一个成员函数, 或者至少有一个用户定义类型的参数(重新定义运算符`new`和`delete`的函数则没有此项要求)。这一规则就保证了用户不能改变原有表达式的意义, 除非表达式中包含有用户定义类型的对象。特别地, 不能定义只对指针进行操作的运算符函数。这些就保证了C++是可扩充的, 但却不是变化无常的(除了针对类对象的运算符`=`、`&`和`,`)。

如果某个运算符函数想接受某个内部类型(4.1.1节)作为第一个参数, 那么它自然就不可能是成员函数了。举个例子, 考虑加一个复数变量到整数2上: 只要有适当声明的成员函数, `aa + 2`可以解释为`aa.operator+(2)`; 但`2 + aa`却不能, 因为没有类`int`能定义`+`去表示`2.operator+(aa)`。即使真能那样做, 我们也将需要用两个不同的成员函数分别去处理`2 + aa`和`aa + 2`。由于编译器不知道用户定义的`+`的意义, 它没有办法保证其可交换性, 不能将`2 + aa`解释为`aa + 2`。这个例子可以很方便地用非成员函数处理(11.3.2节、11.5节)。

枚举也是用户定义类型, 因此可以为它们定义运算符。例如,


```
enum Day { sun, mon, tue, wed, thu, fri, sat };
Day& operator++ (Day& d)
{
    return d = (sat==d) ? sun : Day(d+1);
}
```

每个表达式都需要检查歧义性。如果某个用户定义运算符提供了一个可能的解释，那么就需要根据7.4节的规则去检查表达式。

11.2.4 名字空间里的运算符

一个运算符或者是一个成员函数，或者是定义在某个名字空间里（也可以是全局名字空间）。考虑下面这个标准库串I/O的简化版本

```
namespace std {           // 简化的std
    class ostream {
        // ...
        ostream& operator<<(const char*);
    };
    extern ostream cout;
    class string {
        // ...
    };
    ostream& operator<<(ostream&, const string&);
}

int main()
{
    char* p = "Hello";
    std::string s = "world";
    std::cout << p << ", " << s << "!\n";
}
```

当然，这当然是想写出*Hello, world!* 但为什么呢？注意，我并没有写如下的代码使*std*里的所有东西都可以访问

```
using namespace std;
```

相反，我对*string*和*cout*采用了*std*前缀。换句话说，我按照最好的方式行事，没去污染全局名字空间，也没以其他方式引进不必要的依赖性。

针对C语言风格的字符串（*char**）的输出运算符也是*std::ostream*的一个成员，所以，按定义

```
std::cout << p
```

的意思是

```
std::cout.operator<<(p)
```

然而*std::ostream*本身并没有输出*std::string*的成员函数，所以

```
std::cout << s
```

的意思就是

```
operator<<(std::cout, s)
```

在名字空间里定义的运算符将基于其运算对象的类型去查找，就像基于参数的类型去查找函数一样（8.2.6节）。特别地，因为`cout`在名字空间`std`里，所以在寻找`<<`的适当定义时需要考虑`std`。按照这种方式，编译器发现并使用了

```
std::operator<< (std::ostream&, const std::string&)
```

现在考虑二元运算符`@`，如果`x`的类型为`X`而`y`的类型是`Y`，`x@y`将按如下方式解析：

- 若`X`是类，查寻作为`X`的成员函数或者`X`的某个基类的成员函数的`operator@`。
- 在围绕`x@y`的环境中查寻`operator@`的声明。
- 若`X`在名字空间`N`里定义，在`N`里查寻`operator@`的声明。
- 若`Y`在名字空间`M`里定义，在`M`里查寻`operator@`的声明。

有可能一下找到了`operator@`的若干个声明，如果有的话，这时就用重载解析规则去找出其中的最佳匹配。应用这种查寻机制的条件是至少有一个运算对象是用户定义类型的。为此还需要考虑用户定义的转换（11.3.2节、11.4节）。注意，`typedef`名字只是同义词，而不是用户定义类型（4.9.7节）。

一元运算符的解析方式与此类似。

请注意，运算符查寻机制并不认为成员函数比非成员函数更应优先选取。这一点与命名函数的查寻规则不同（8.2.6节）。不存在运算符屏蔽的问题，这也就保证了内部运算符绝不会变得无法使用；也保证用户可以为运算符提供新的意义，而不必去修改现存的类声明。例如，标准`iostream`库为内部类型定义了`<<`成员函数。用户可以定义`<<`以输出用户定义类型，不必去修改`ostream`类（21.2.1节）。

11.3 一个复数类型

在前面引言部分给出的复数类过于简单，任何人都不会喜欢它。举例来说，查看一本数学手册，我们可能希望下面这些都能做：

```
void f()
{
    complex a = complex(1, 2);
    complex b = 3;
    complex c = a+2.3;
    complex d = 2+b;
    complex e = -b-c;
    b = c*2*c;
}
```

此外，我们还可能希望提供另外一些运算符，例如以`==`做比较，`<<`做输出，还有一组适当的数学函数，如`sin()`和`sqrt()`等。

类`complex`是一个具体类型，所以它的设计将沿着10.3节指明的方向进行。另外，复数类型的用户对各种运算符的依赖性如此之强，因此，`complex`的定义将使有关运算符重载的大多数基本规则都能有所表现。

11.3.1 成员运算符和非成员运算符

我喜欢让尽量少的函数能直接去操作对象的表示。为此，可以只在类本身之中定义那些

本质上就需要修改其第一个参数值的运算符，如 `+=`。而像 `+` 这样基于参数的值简单产生新值的运算符，则可以在类之外利用类实现中的基本运算符来定义：

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a);    // 需要访问对象表示
    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b;    // 通过 += 访问表示
}
```

有了这些定义，我们就可以写：

```
void f(complex x, complex y, complex z)
{
    complex r1 = x+y+z;    // r1 = operator + (operator + (x,y),z)
    complex r2 = x;        // r2 = x
    r2 += y;                // r2.operator += (y)
    r2 += z;                // r2.operator += (z)
}
```

除了可能有性能差异外，`r1`和`r2`的计算是完全等价的。

如 `+=` 和 `*=` 这样的组合赋值运算符，比它们“貌似简单”的对应物 `+` 和 `*` 定义起来更简单。这一情况在一开始使许多人感到吃惊。其实这是一个事实造成的：`+` 运算符涉及到3个对象（两个运算对象和一个结果），而在 `+=` 运算中只涉及到两个对象。对于后一个运算，通过清除对临时变量的需要，可以改进运行效率。例如，

```
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

这样就不需要临时变量来保存加的结果，也使编译器很容易实施完美的在线化。

一个好的优化程序可以对普通 `+` 运算符的使用生成近乎最优的代码。但是，由于我们未必总有好的优化程序，也不是所有类型都像 `complex` 这样简单，所以，在11.5节里，我们还将讨论采用直接访问类表示的方式定义运算符的种种方法。

11.3.2 混合模式算术

为了处理

```
complex d = 2+b;
```

我们需要定义能够接受不同类型的运算对象的 `+` 运算符。采用Fortran的术语，我们需要混合模式算术。我们当然可以通过简单地增加适当的运算符版本的方式达到这个目标：

```
class complex {
    double re, im;
```

```

public:
    complex& operator+=(complex a) {
        re += a.re;
        im += a.im;
        return *this;
    }

    complex& operator+=(double a) {
        re += a;
        return *this;
    }

    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b; // 调用 complex::operator += (complex)
}

complex operator+(complex a, double b)
{
    complex r = a;
    return r += b; // 调用 complex::operator += (double)
}

complex operator+(double a, complex b)
{
    complex r = b;
    return r += a; // 调用 complex::operator += (double)
}

```

将一个`double`加到一个复数上，是比加上一个`complex`更简单的运算，上述定义里也反映出这种情况。以`double`为运算对象的运算符并没有触及复数的虚部，因此可能会更有效。

有了这些定义，我们就可以写

```

void f(complex x, complex y)
{
    complex r1 = x+y; // 调用 operator + (complex,complex)
    complex r2 = x+2; // 调用 operator + (complex,double)
    complex r3 = 2+x; // 调用 operator + (double,complex)
}

```

11.3.3 初始化

要想用标量来对`complex`变量做初始化和赋值，我们就需要从标量（整数或者浮点数）到`complex`的转换。例如，

```
complex b = 3; // 应该表示 b.re = 3, b.im = 0
```

具有一个参数的构造函数就刻画了由其参数类型到它构造起的类型的转换。例如，

```

class complex {
    double re, im;
public:
    complex(double r) : re(r), im(0) { }
    // ...
};

```

这个构造函数描述的是实数轴在复平面上的传统嵌入。

构造函数是有关如何建立起给定类型的一个值的处方。当程序里需要一个某类型的值，而某个构造函数又能通过把所提供的值作为初始式或者被赋的值，去创建起一个这样的值的时候，这个构造函数就会被调用。因此，具有一个参数的构造函数也可能不需要显式地调用。例如，

```
complex b = 3;
```

的意思是

```
complex b = complex(3);
```

只有在某个用户定义转换具有惟一性（7.4节）时，它才会被隐式地调用。参看11.7.1节有关如何定义只能显式地调用的构造函数的方法。

很自然，我们也需要带有两个`double`参数的构造函数，将`complex`初始化为 $(0, 0)$ 的默认构造函数也非常有用：

```
class complex {
    double re, im;
public:
    complex() : re(0), im(0) { }
    complex(double r) : re(r), im(0) { }
    complex(double r, double i) : re(r), im(i) { }
    // ...
};
```

利用默认参数机制，我们就可以有下面简写：

```
class complex {
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) { }
    // ...
};
```

如果对某个类型显式地声明了构造函数，那么就不能再用初始式的列表（5.7节、4.9.5节）作为变量的初始式了。例如，

```
complex z1 = { 3 };      // 错误：complex有构造函数
complex z2 = { 3, 4 };   // 错误：complex有构造函数
```

11.3.4 复制

除了显式声明的构造函数之外，`complex`还按照默认规定，得到了一个定义好的复制构造函数（10.2.5节）。默认的复制构造函数就是简单地复制成员，为使其更明显，我们也可以等价地写：

```
class complex {
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) { }
    // ...
};
```

当然，对于那些默认的复制构造函数正好具有正确语义的类型，我更喜欢依靠默认定义。这样做比我自己写少说一些话，读者也应理解这种默认规定。还有，编译器知道这种默认方式，因此就有机会做一些可能的优化。进一步说，用手写出逐个成员的复制也是很烦人的事情，对于有很多数据成员的类型也容易出错（10.4.6.3节）。

我对复制构造函数采用的是引用参数，这样做是必须的。复制构造函数定义了复制的意义——包括复制参数的意义——所以，写

```
complex::complex(complex c) : re(c.re), im(c.im) {} // 错误
```

是个错误，因为这将会使任何调用都陷入无穷的递归。

至于其他以`complex`为参数的函数，我用的都是值参数而不是引用参数。在这里设计者有一个选择。从用户的观点看，一个函数是采用`complex`参数还是 `const complex&` 参数并没有什么不同。这个问题将在11.6节里进一步讨论。

从原则上说，复制构造函数也用在简单的初始化中，如

```
complex x = 2;           // 建立complex(2)，而后用它初始化x
complex y = complex(2, 0); // 建立complex(2, 0)，而后用它初始化y
```

但是，对复制构造函数的调用很容易通过优化去掉。我们可以等价地写

```
complex x(2);           // 用2初始化x
complex y(2, 0);        // 用(2, 0)初始化y
```

对于类似`complex`这样的算术类型，我认为使用 `=` 的形式看起来更好些。我们可以对采用 `=` 风格的初始化所能接受的值集合加以限制（与采用 `()` 形式的初始化相比），方式是将复制构造函数做成私用的（11.2.2节），或者是将某个构造函数声明为`explicit`（11.7.1节）。

与初始化类似，同属一个类的两个对象之间的赋值也默认地定义为按成员赋值（10.2.5节）。我们可以显式地定义`complex::operator =` 去做这件事。当然，对于像`complex`这样的简单类型，完全没有理由去那样做，因为默认的东西正好合适。

复制构造函数——无论是用户定义的还是编译器生成的——不但被用在初始化变量时，也用在参数传递、值返回，以及异常处理中（11.7节）。这些操作的语义都被定义成初始化的语义（7.1节、7.3节、14.2.1节）。

11.3.5 构造函数和转换

我们为每个标准的算术运算定义了三个版本

```
complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);
// ..
```

这可能成为令人厌倦的事情，而厌倦很容易变成错误的根源。如果函数的每个参数有三种不同情况，那又会怎么样？对于一个参数的函数，我们就需要3个不同版本，两个参数的函数需要9个不同版本，三个参数的函数是27个不同版本，如此等等。通常这些版本都非常类似，事实上，几乎所有的版本都涉及到一个从参数类型到公共类型的简单转换，而后是一个标准的算法。

与为参数的不同组合分别提供函数版本相对应的还有另外一种方式，那就是依赖于类型

转换。例如，我们的`complex`类提供了一个构造函数，它能将`double`转换到`complex`。由于这种情况，我们就只需要为`complex`的等于运算符定义一个版本

```
bool operator==(complex, complex);

void f(complex x, complex y)
{
    x==y;    // 意思是operator==(x,y)
    x==3;    // 意思是operator==(x,complex(3))
    3==y;    // 意思是operator==(complex(3),y)
}
```

也存在一些使人倾向于定义多个不同版本的因素。例如，在一些情况下转换将带来额外的开销；另一些情况中，对于某个特定参数类型可以使用更简单的算法，如此等等。如果在某个地方这些事情都不重要，那就可以依赖于转换，只提供函数的一个最通用的形式——也可能加上若干关键的变形——以遏制由于混合模式算术引起的不同变体的组合爆炸问题。

在那种存在一个函数或者运算符的多个版本的地方，编译器必须基于参数的类型和可用的（标准的和用户定义的）转换，选取出一个“正确的”版本。除非确实存在一个最佳匹配，否则这个表达式就是有歧义的，是一个错误（见7.4节）。

在一个表达式里，通过显式或者隐式地使用构造函数创建的对象是自动对象，它们将在最早可能的时刻被销毁（见10.4.10节）。

任何用户定义的转换都不会用到，（或者 \rightarrow ）的左边。即使在，是隐含的情况下也同样如此。例如，

```
void g(complex z)
{
    3+z;           // 可以: complex(3)+z
    3.operator+=(z); // 错误: 3不是类对象
    3+=z;          // 错误: 3不是类对象
}
```

由此，你可以表达某个运算符要求其左边运算对象是左值这样的概念，为此只需要将该运算符作为成员函数。

11.3.6 文字量

不能为类类型定义文字量（像对`double`类型的文字量`1.2`或`12e3`那样）。但无论如何，我们常常可以利用内部类型的文字量，只需要通过类的成员函数为其提供一种解释。只有一个参数的构造函数就是完成这种事情的机制。当构造函数很简单并且被在线化时，把这种以文字量为参数的构造函数调用就看成是文字量，这种看法也相当合理。例如，我就把`complex(3)`看做是`complex`类型的文字量，虽然从技术上说它确实不是。

11.3.7 另一些成员函数

至此，我们还只为`complex`类提供了构造函数和算术运算符，对于实际应用而言，这些当然还不够。特别地，我们常常需要去检查实部和虚部的值：

```
class complex {
    double re, im;
public:
```

```

    double real() const { return re; }
    double imag() const { return im; }
    // ...
};

```

与`complex`的其他成员不同，`real()`和`imag()`并不修改`complex`的值，因此可以将它们定义为`const`。

有了`real()`和`imag()`，我们就能定义各种各样的有用运算，而不需要让这些运算去直接访问`complex`的内部表示。例如，

```

inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}

```

注意，我们只需要去读实部和虚部，改写是更不常见的操作。如果我们必须做“部分更新”，那也可以做：

```

void f(complex& z, double d)
{
    // ...
    z = complex(z.real(), d); // 将d赋给z.im
}

```

一个好的编译器应该只对这个语句生成一个赋值。

11.3.8 协助函数

把所有的东西集中到一起，`complex`类就变成了：

```

class complex {
    double re, im;
public:
    complex(double r=0, double i=0) : re(r), im(i) {}

    double real() const { return re; }
    double imag() const { return im; }

    complex& operator+=(complex);
    complex& operator+=(double);
    // -=, *=, 和 /=
};

```

除此之外，我们还必须提供一批协助函数：

```

complex operator+(complex, complex);
complex operator+(complex, double);
complex operator+(double, complex);

// -, *, 和 /

complex operator-(complex); // 一元负号
complex operator+(complex); // 一元正号

bool operator==(complex, complex);
bool operator!=(complex, complex);

istream& operator>>(istream&, complex&); // 输入
ostream& operator<<(ostream&, complex&); // 输出

```


请注意, 成员函数`real()`和`imag()`是定义各种比较运算的基础。下面的许多协助函数的定义也依赖于`real()`和`imag()`。

我们还可能提供一些函数, 使用户可以按照极坐标的方式考虑复数的问题:

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);    // 记法转换
double imag(complex);    // 记法转换
```

最后, 我们还需要提供一组适当的标准数学函数:

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

从用户的观点看, 这里给出的复数类型基本上等同于可以在标准库的`<complex>`里找到的`complex<double>` (22.5节)。

11.4 转换运算符

通过构造函数去刻画类型转换确实很方便, 但这也意味着一些我们可能并不希望的情况。构造函数不能刻画

[1] 从用户定义类型到一个内部类型的转换 (因为内部类型不是类)。

[2] 从新类型到某个已有类型的转换 (而不去修改那个已有类的声明)。

这些问题都可以通过为源类型定义转换运算符的方式解决。成员函数`X::operator T()`, 其中的`T`是一个类型名, 就定义了一个从`X`到`T`的转换。例如, 可以定义一种包含6个二进制位的非负整数`Tiny`, 并允许它自由地与整数做各种算术运算:

```
class Tiny {
    char v;
    void assign(int i) { if (i < 0) throw Bad_range(); v=i; }
public:
    class Bad_range { };

    Tiny(int i) { assign(i); }
    Tiny& operator=(int i) { assign(i); return *this; }

    operator int() const { return v; }    // 转换到int的函数
};
```

在所有用`int`来初始化一个`Tiny`, 或者用`int`给`Tiny`赋值的地方都需要做值范围检查。而在需要做`Tiny`的复制时就不必检查了, 所以默认的复制构造函数和赋值都恰好是正确的。

为了能对`Tiny`变量使用普通整数运算, 我们在这里定义了从`Tiny`到`int`的隐式转换`Tiny::operator int()`。请注意, 作为转换目标的类型是运算符名字的一部分, 不能作为转换函数的返回值类型而重复写出:

```
Tiny::operator int() const { return v; }    // 正确
int Tiny::operator int() const { return v; } // 错误
```

也正是从这个角度看，转换运算符很像是构造函数。

如果*Tiny*出现在任何需要*int*的地方，就会使用适当的*int*。例如，

```
int main()
{
    Tiny c1 = 2;
    Tiny c2 = 62;
    Tiny c3 = c2 - c1;    // c3 = 60
    Tiny c4 = c3;        // 不检查值域（不必要）
    int i = c1 + c2;      // i = 64

    c1 = c1 + c2;        // 值域错误：c1不能是64
    i = c3 - 64;         // i = -4
    c2 = c3 - 64;        // 值域错误：c2不能是-4
    c3 = c4;             // 不检查值域（不必要）
}
```

在某种数据结构的读出操作（通过转换运算符实现）非常简单，而赋值和初始化又不那么简单，在处理这种数据结构时，转换函数似乎特别有用。

*istream*和*ostream*类型也依靠转换函数，这使我们可以写下面这类语句：

```
while (cin >> x) cout << x;
```

输入操作`cin >> x`返回*istream*&，这个值将被隐式地转换到一个表明*cin*状态的值，而这个值将由*while*检测（见21.3.3节）。当然，通常定义这样的转换并不是一个好主意，以这种方式从一个类型隐式地转换到另一个，并在转换中丢失了信息^①。

总而言之，少定义一些转换运算符是更明智的。如果用得过分，这种东西很容易引起歧义性。编译器能捕捉到这样的歧义性，但解决它们有时也会成为很困难的问题。最好的方式或许是在一开始先用命名函数实现转换，例如写`X::make_int()`。当这种函数出现的太频繁，显式使用变得很不美观时，再用一个转换运算符`X::operator int()`取代之。

如果同时存在用户定义转换和用户定义运算符，那么也可能产生用户定义运算符和内部运算符之间的歧义性问题。例如，

```
int operator+ (Tiny, Tiny);
void f(Tiny t, int i)
{
    t+i; // 错误，歧义：operator+(t, Tiny(i)) 或 int(t)+i?
}
```

所以，对于一个类型而言，最好是或者依靠用户定义转换，或者依靠用户定义运算符，但不要两者都用。

11.4.1 歧义性

如果存在一个赋值运算符`X::operator= (Z)`，使得*V*就是*Z*，或者存在着惟一的一个从*V*到*Z*的转换，以类型*V*的值为类*X*的对象赋值是合法的。初始化的处理与此等价。

① 在这个实际例子里，出现的转换是从*istream*到*bool*的隐式转换，转换的结果正好可以被*while*语句用于检查输入流的状态。但在另一方面，这个转换中却将原来的流丢掉了（由转换得到的*bool*值当然无法确定原来的流），因此说，这是一个“丢失信息的转换”。作者这段话的意思就是：如果不是因为特别的需要，一定不要定义“丢失信息的转换”。——译者注

也存在一些情况,那里所需类型的值可以通过反复应用构造函数或转换运算符构造出来。在这时就必须通过显式的转换处理,能合法进行的用户定义隐式转换只有一层。还可能有一种情况,其中所需的某个类型的值可以通过多种途径构造起来,这种情况就是有歧义的。例如,

```
class X { /* ... */ X(int); X(char*); };
class Y { /* ... */ Y(int); };
class Z { /* ... */ Z(X); };

X f(X);
Y f(Y);

Z g(Z);
void kl()
{
    f(I);           // 错误: 歧义的f(X(I)) 或 f(Y(I)) ?
    f(X(I));        // ok
    f(Y(I));        // ok

    g("Mack");      // 错误: 需要两次用户定义转换; 不会试验 g(Z(X("Mack")))
    g(X("Doc"));    // ok: g(Z(X("Doc")))
    g(Z("Suzy"));   // ok: g(Z(X("Suzy")))
```

只有在解析一个调用时有需要,才会去考虑用户定义的转换。例如,

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX);

void k2()
{
    h(I);           // h(double(I))或h(XX(I))? h(double(I))!
```

调用`h(I)`的意思是`h(double(I))`,因为在这一选择中只使用了标准转换,不需要任何用户定义的转换(7.4节)。

有关转换的规则既不是最容易实现的,也不是最容易写出文档的,通常也无法达到某种最具一般性的东西。然而,它们是相当安全的,得到的结果也是最不容易出人意料的。与发现一个由于意料之外的转换而导致的错误相比,通过手工方式消解歧义性要容易得多。

强调严格的自下而上分析也意味着在重载解析中不使用返回值类型。例如,

```
class Quad {
public:
    Quad(double);
    // ...
};

Quad operator+(Quad, Quad);

void f(double a1, double a2)
{
    Quad r1 = a1+a2;           // 双精度加
    Quad r2 = Quad(a1)+a2;     // 强制要求Quad算术
}
```

做出这种设计选择的原因，一方面是认为严格的自下而上分析比较容易理解，此外就是认为，确定程序员可能希望哪种加法不应该是编译器的工作。

一旦某个初始式或者赋值两边的类型都确定了，就利用这两个类型去做初始化或者赋值的解析。例如，

```
class Real {
public:
    operator double();
    operator int();
    // ...
};

void g(Real a)
{
    double d = a;    // d = a.double();
    int i = a;       // i = a.int();

    d = a;           // d = a.double();
    i = a;           // i = a.int();
}
```

对于所有这些情况，类型分析都是自下而上地进行，每次只考虑一个运算符和它的参数类型。

11.5 友元

一个常规的成员函数声明描述了三件在逻辑上相互不同的事情：

- [1] 该函数能访问类声明的私用部分。
- [2] 该函数位于类的作用域之中。
- [3] 该函数必须经由一个对象去激活（有一个`this`指针）。

通过将函数声明为`static`（10.2.4节），我们可以让它只具有前两种性质。通过将一个函数声明为友元（`friend`），可以使它只具有第一种性质。

例如，我们可以定义一个运算符去做`Matrix`和`Vector`的乘法。当然，`Matrix`和`Vector`都会将它们的表示隐蔽起来，并提供一组操作它们类型的对象的运算。然而，我们所要的这个乘法例程不可能同时成为两者的成员函数。此外，我们可能也不希望提供一些低级访问函数，使每个用户都能对`Matrix`和`Vector`的完整表示进行读写操作。为避免这些情况，我们可以声明这个`operator*`作为两个类的友元：

```
class Matrix;

class Vector {
    float v[4];
    // ...
    friend Vector operator* (const Matrix&, const Vector&);
};

class Matrix {
    Vector v[4];
    // ...
    friend Vector operator* (const Matrix&, const Vector&);
};

Vector operator* (const Matrix& m, const Vector& v)
{
```

```

    Vector r;
    for (int i = 0; i < 4; i++) {        // r[i] = m[i] * v;
        r.v[i] = 0;
        for (int j = 0; j < 4; j++) r.v[i] += m.v[i].v[j] * v.v[j];
    }
    return r;
}

```

friend声明可以放在类声明中的私用部分或者公用部分，放在哪里都没关系。与成员函数一样，友元函数也是在将它视为友元的类中明确声明的，因此，它也将像成员函数一样成为类界面的一部分。

一个类的成员函数也可以是另一个类的友元。例如，

```

class List_iterator {
    // ...
    int* next();
};

class List {
    friend int* List_iterator::next();
    // ...
};

```

常有这种情况，某个类的所有成员函数都需要作为另一个类的友元。下面是对这种情况的简写形式：

```

class List {
    friend class List_iterator;
    // ...
};

```

这个友元声明就使 *List_iterator* 的成员函数都成为 *List* 的友元了。

易见，**friend**类只应该用于那些密切相关的概念。常常会需要做出选择，是将一个类作为成员（嵌套类）呢，还是将它作为非成员的友元（24.4节）。

11.5.1 友元的寻找

像成员的声明一样，一个友元声明不会给外围的作用域引进一个名字。例如，

```

class Matrix {
    friend class Xform;
    friend Matrix invert(const Matrix&);
    // ...
};

Xform x;                                // 错误：作用域里无Xform
Matrix (*p)(const Matrix&) = &invert;  // 错误：作用域里无invert()

```

对于大型程序和大的类，一个类不能“默不作声地”给它的外围作用域加入一些名字，维持这种性质是非常好的事情。对于能够在许多不同环境里实例化的模板（第13章）而言，这一规定就特别重要。

一个友元类必须或者是在外围作用域里先行声明的，或者是在将它作为友元的那个类的直接外围的非类作用域里定义的。在外围的最内层名字空间作用域之外的作用域不在考虑之列。例如，

```

class X { /* ... */ };           // Y的友元
namespace N {
    class Y {
        friend class X;
        friend class Z;
        friend class AE;
    };
    class Z { /* ... */ };       // Y的友元
}
class AE { /* ... */ };         // 不是Y的友元

```

友元函数可以像友元类一样明确地声明。此外，还可以通过它的参数找到（8.2.6节），即使如果它并没有在外围最近的作用域里声明。例如，

```

void f(Matrix& m)
{
    invert(m);    // Matrix的友元invert()
}

```

由此，一个友元函数或者需要在某个外围作用域里显式声明，或者以它的类或由该类派生的类作为一个参数（13.6节），否则就无法调用这个友元了。例如，

```

// 假定作用域里无f()
class X {
    friend void f();           // 无用
    friend void h(const X&); // 可以通过参数找到
};

void g(const X& x)
{
    f();    // 作用域里无f()
    h(x);   // 调用X的友元h()
}

```

11.5.2 友元和成员

我们什么时候应该选用友元函数，什么时候用成员函数刻画一个运算才是更好的选择呢？首先，我们应设法尽可能地减少访问类表示的函数的数目，并尽可能地将能访问的函数集合做好。因此，第一个问题并不是“它应该是成员、*static*成员还是友元？”而是“它确实需要做这种访问吗？”典型的情况是，需要做这种访问的函数集合比我们一开始所认为的更小一些。

有些操作必须是成员——例如，构造函数、析构函数和虚函数（12.2.6节）——但对一般的函数常常可以有所选择。由于成员名是局部于类的，这些函数还是应该优先被作为成员，除非存在某个特殊原因说它应该不是成员。

考虑下面的类X，其中显示了提供操作的各种不同方式：

```

class X {
    // ...
    X(int);

    int m1();
    int m2() const;
}

```

```

    friend int f1(X&);
    friend int f2(const X&);
    friend int f3(X);
};

```

成员函数只能通过有关类的对象去调用，在这里不会使用用户定义转换。例如，

```

void g()
{
    99.m1(); // 错误：不会试X(99).m1()
    99.m2(); // 错误：不会试X(99).m2()
}

```

这里不会应用 $X(int)$ ，不会从99做出一个X来。

全局函数 $f1()$ 具有类似的性质，因为隐式转换不会被用于非`const`的引用参数（5.5节、11.3.5节）。然而，对于 $f2()$ 和 $f3()$ 的参数就可以实施转换：

```

void h()
{
    f1(99); // 错误：不会试f1(X(99))
    f2(99); // ok: f2(X(99));
    f3(99); // ok: f3(X(99));
}

```

由此可见，一个修改类对象状态的操作应该或者是一个成员，或者是一个带有非`const`引用参数（或非`const`指针参数）的全局函数。那些要求基础类型的左值运算对象的运算符（`=`、`*=`、`++`等）作为用户定义类型的成员函数最为自然。

与此相反，如果希望某个运算的所有运算对象都能允许隐式类型转换，实现它的函数就应该作为非成员函数，取`const`引用参数或者非引用参数。那些在应用时不需要基础类型的左值的运算符（`+`、`-`、`||`等），实现它们的函数常常采用这种方式。这些运算符经常需要访问其运算对象类的内部表示，因此，它们是`friend`函数的最常见的来源。

如果没有定义类型转换，那么看起来就不存在很有效的理由，说明选择成员函数比采用引用参数的友元函数更好，或者反过来。在某些情况下，程序员可能更偏爱某一种调用语法的形式而胜过另一种。例如，大部分人似乎更喜欢用记法形式 $inv(m)$ 表示求`Matrix m`的转置，而不是另一种可能形式 $m.inv()$ 。当然，如果 $inv()$ 所做的真是 m 本身的转置，而不是返回一个新的`Matrix`作为 m 的转置，那么它就应该是成员了。

如果在所有应该考虑的方面都难分伯仲，那么就选择成员。我们无法知道是否在某人某日将需要定义一个转换运算符，也不可能预料到将来的某个改动是否需要修改所涉及对象的状态。成员函数的调用语法使用户非常清楚：这个对象可能被修改；而引用参数就远没有那么明显。进一步说，在成员函数体里的表达式比全局函数体内等价的表达式短许多，因为成员可以隐式地利用`this`。还有，成员的名字局部于类，它们通常也比非成员函数的名字更短一些。

11.6 大型对象

我们将`complex`的运算符都定义为以类型`complex`为参数，这也就意味着，对于`complex`的每个运算符，各个运算对象都需要复制。复制两个`double`的开销或许可以察觉到，但却常常小于采用一个指针所导致的开销（相对而言，通过指针访问的代价可能会更大一些）。然而，并不是所有的类都有某种很方便的很小的表示方式。为避免过度的复制，我们就需要定义以

引用为参数的函数。例如，

```
class Matrix {
    double m[4][4];
public:
    Matrix();
    friend Matrix operator+(const Matrix&, const Matrix&);
    friend Matrix operator*(const Matrix&, const Matrix&);
};
```

引用使我们能对大型对象使用牵涉到各种常见算术运算符的表达式，而又能避免过度的复制。在这里不能用指针，因为不能针对指针重新定义运算符的意义。加法的定义可能是

```
Matrix operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix sum;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
            sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
    return sum;
}
```

这个`operator+`() 通过引用去访问 + 的运算对象，但返回的是对象值。返回一个引用值看起来可能更加有效些：

```
class Matrix {
    // ...
    friend Matrix& operator+(const Matrix&, const Matrix&);
    friend Matrix& operator*(const Matrix&, const Matrix&);
};
```

这样做是合法的，但却会带来存储分配问题。因为一个到结果的引用将被函数作为到返回值的引用传出来，所以，这个返回值就不能是自动变量（7.3节）。因为一个运算符可能在某个表达式里多次使用，因此结果也不能是`static`局部变量。这种结果通常需要在自由空间里分配。复制结果值常常比在自由存储里分配并（最终）释放一个对象更廉价些（从执行时间、代码空间和数据空间看）。这种做法也能使程序简单许多。

也存在一些你可以用于避免复制结果的技术。最简单的就是采用一个静态对象的缓冲区。例如，

```
const int max_matrix_temp = 7;

Matrix& get_matrix_temp()
{
    static int nbuf = 0;
    static Matrix buf[max_matrix_temp];

    if (nbuf == max_matrix_temp) nbuf = 0;
    return buf[nbuf++];
}

Matrix& operator+(const Matrix& arg1, const Matrix& arg2)
{
    Matrix& res = get_matrix_temp();
    // ...
    return res;
}
```


现在，只有在表达式结果的赋值时才需要复制`Matrix`。当然，如果你写出一个表达式，其中涉及到的临时量多于`max_matrix_temp`，只有老天爷才能帮你的忙！

另一种不容易出错的技术涉及到将矩阵类型定义为一种句柄（25.7节），让它关联于另一个实际保存数据的表示类型。在这种方式下，矩阵句柄可以按一种可以只做最少的分配和复制的方式去管理相应的表示对象（11.2节和11.14[18]）。然而，这种策略的基础是让运算符返回对象，而不是返回引用或者指针。另一种技术是定义一些三元运算，并让它们能够在遇到如`a = b + c`和`a + b * i`一类表达式时被自动调用（21.4.6.3节、22.4.7节）。

11.7 基本运算符

一般说来，对于类型`X`，复制构造函数`X(const X&)`负责完成从同类型`X`的一个对象出发的初始化。强调赋值和初始化是不同操作绝不会过分（10.4.4.1节）。当声明了一个析构函数时，这一点尤为重要。如果类`X`有一个完成非平凡工作的析构函数，例如它需要释放自由存储，这个类就很可能需要一组完整的互为补充的函数来控制构造、析构和复制：

```
class X {
    // ...
    X(Sometype);           // 构造函数：创建对象
    X(const X&);           // 复制构造函数
    X& operator=(const X&); // 复制赋值：清理和复制
    ~X();                  // 析构函数：清理
};
```

还存在另外三种需要复制对象的情况：将对象作为函数的参数，作为函数的返回值，作为一个异常。在传递参数时，将对那个当时尚未初始化的变量——形式参数——进行初始化，在语义上与其他初始化完全一样。对于函数返回值和异常，情况也相同，虽然它们不那么明显。对这些情况，都需要使用复制构造函数。例如，

```
string g(string arg)      // string作为值传递（用复制构造函数）
{
    return arg;           // 返回string（用复制构造函数）
}

int main ()
{
    string s = "Newton";   // 初始化string（用复制构造函数）
    s = g(s);
}
```

很清楚，在调用`g()`之后`s`的值应该是`"Newton"`。取得`s`值的一个副本给参数`arg`并不困难，调用`string`的复制构造函数就能完成这件事情。要取得这个值的另一个副本送出`g()`，则需要再次调用`string(const string&)`，这次被初始化的变量是一个临时对象（10.4.10节），而后将它给`s`赋值。在这两次复制中常常有一个能够通过优化去掉，不可能两个都去掉。

对于一个类，如果程序员没有显式地声明复制赋值或者复制构造函数，编译器将会生成所缺少的操作（10.2.5节）。这也意味着复制操作是不会继承的（12.2.3节）。

11.7.1 显式构造函数

按默认规定，只有一个参数的构造函数也定义了一个隐式转换。对于某些类型，这一情

况就非常理想。例如，可以用`int`为一个`complex`做初始化：

```
complex z = 2; // 用complex(2) 初始化z
```

但在另一些情况中，隐式转换却是我们所不希望的，容易导致错误。例如，如果我们可以用一个`int`作为大小去初始化一个`string`，某人可能写出

```
string s = 'a'; // 将s做成一个包含int('a') 个元素的string
```

很难认为这会是人们定义`s`时所希望的意思。

通过将构造函数声明为`explicit`（显式）的方式就可以抑制隐式转换。也就是说，`explicit`构造函数必须显式调用。特别地，在那些原则上需要复制构造函数的地方（11.3.4节），将不会被隐式地调用`explicit`构造函数。例如，

```
class String {
    // ...
    explicit String(int n);    // 预先分配n个字节
    String(const char* p);    // 用C风格的字符串p作为初始值
};

String s1 = 'a';             // 错误：不能做隐式char -> String转换
String s2(10);               // 可以：10个字符的String
String s3 = String(10);      // 可以：10个字符的String
String s4 = "Brian";         // 可以：s4 = String("Brian")
String s5("Fawlty");

void f(String);

String g()
{
    f(10);                   // 错误：不能做隐式int -> String转换
    f(String(10));
    f("Arthur");             // ok: f(string("Arthur"))
    f(s1);

    String* p1 = new String("Eric");
    String* p2 = new String(10);

    return 10;               // 错误：不能做隐式int -> String转换
}
```

有些差异看起来有点做作，如在

```
String s1 = 'a';             // 错误：不能做隐式char -> String转换
```

和

```
String s2(10);               // 可以：10个字符的String
```

之间。在实际代码中的东西则不像这种故意造出的例子。

在`Date`里，我们用普通的`int`表示年份（10.3节）。如果`Date`在我们的设计中极其重要，那么我们会引进一个`Year`类型，使编译器能去做更强的检查。例如，

```
class Year {
    int y;
public:
    explicit Year(int i) : y(i) { }    // 从int创建Year
    operator int() const { return y; } // 转换：从Year到int
};
```

```

class Date {
public:
    Date(int d, Month m, Year y);
    // ...
};

Date d3(1978, feb, 21); // 错误: 21不是Year
Date d4(21, feb, Year(1978)); // ok

```

类`Year`不过是包裹住`int`的一层布。由于有`operator int()`，只要需要，一个`Year`就可以隐式地转换到`int`。通过将构造函数声明为`explicit`，我们就能保证，`int`到`Year`的转换只能在明确要求的方进行，而那些“意外的”赋值将在编译时被捕捉到。由于`Year`的成员函数很容易在线化，这样做不会增加时间或者空间开销。

类似技术也可用于定义表示范围的类型（25.6.1节）。

11.8 下标

函数`operator[]`可以用于为类的对象定义下标运算的意义。`operator[]`的第二个参数（下标）可以具有任何类型，这就使我们可以去定义`vector`、关联数组等。

作为一个例子，让我们重新做5.5节的实例。在那里我们用一个关联数组，写出了一个很小的统计在文件中单词出现次数的程序。在那里用的是一个函数，现在要定义一个关联数组类型

```

class Assoc {
    struct Pair {
        string name;
        double val;
        Pair(string n = "", double v = 0) : name(n), val(v) {}
    };
    vector<Pair> vec;

    Assoc(const Assoc&); // 私用，防止复制
    Assoc& operator=(const Assoc&); // 私用，防止复制
public:
    Assoc() {}
    const double& operator[](const string&);
    double& operator[](string&);
    void print_all() const;
};

```

在`Assoc`里保存着一个`Pair`的向量。这个实现采用的仍是与5.5节里一样的很简单的和效率较低的检索方式：

```

double& Assoc::operator[](string& s)
// 检索s；如果找到就返回其值；否则，做一个新的Pair并返回默认值0
{
    for (vector<Pair>::const_iterator p = vec.begin(); p != vec.end(); ++p)
        if (s == p->name) return p->val;

    vec.push_back(Pair(s, 0)); // 初始值: 0
    return vec.back().val; // 返回最后元素（16.3.3节）
}

```

由于`Assoc`的内部表示是隐蔽的，我们需要有一种打印它的方法：

```
void Assoc::print_all() const
{
    for (vector<Pair>::const_iterator p = vec.begin(); p != vec.end(); ++p)
        cout << p->name << ": " << p->val << '\n';
}
```

最后，我们可以写出一个简单的主程序：

```
int main() // 计算每个单词在输入时出现的次数
{
    string buf;
    Assoc vec;
    while (cin >> buf) vec[buf]++;
    vec.print_all();
}
```

对关联数组这一想法的进一步开发可以在17.4.1节找到。

函数`operator[]()`必须是成员函数。

11.9 函数调用

函数调用，即记法形式`expression(expression-list)`，也可以解释为一种二元运算，其中将`expression`作为左运算对象，而`expression-list`作为右运算对象^①。这一调用运算符`()`也可以采用像其他运算符一样的重载方式。`operator()()`的参数列表将被求值，并按普通的参数传递规则进行检查。重载的函数调用看来是非常有用的东西，特别是对定义那些只有一个运算的类型，和那些具有某个主导运算的类型。调用运算符也被称为应用运算符。

运算符`()`的最明显、或许也是最重要的应用是为对象提供常规的函数调用语法形式，使它们具有像函数似的行为方式。一个活动起来像函数的对象常常被称做一个拟函数对象，或简称为函数对象（18.4节）。这种函数对象非常重要，我们可以借助于它们去写出以很不简单的操作作为参数的代码。例如，标准库提供了许多算法，其中要对容器里的每个元素调用一个操作。考虑

```
void negate(complex& c) { c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(), aa.end(), negate); // 对vector中所有元素求负
    for_each(ll.begin(), ll.end(), negate); // 对list中所有元素求负
}
```

这将使向量和表中的每个元素都变成它的负值。

如果我们希望给每个元素加上`complex(2, 3)`又该怎么办？这件事也很容易完成：

```
void add23(complex& c)
{
    c += complex(2, 3);
}

void g(vector<complex>& aa, list<complex>& ll)
```

① 实际上，在表达式表中可以有任意多个表达式，这就使调用运算符成为惟一的一种可以定义任意数目的参数的运算符，而不仅是“二元运算符”。——译者注

```
{
    for_each(aa.begin(), aa.end(), add23);
    for_each(ll.begin(), ll.end(), add23);
}
```

但是，如果我们想写一个函数去重复地加上一个任意的复数值，那又该怎么做呢？我们需要某种东西，可以将某个任意的值传递给它，并在每次调用它的时候使用这个值。用函数来实现这种要求很不自然。典型的方式是，我们最终需要将这个任意的值遗留在函数的调用环境中，以此来“传递”这个值。这种做法极其糟糕。然而，我们可以写一个类，使它恰好具有我们所希望的行为方式：

```
class Add {
    complex val;
public:
    Add(complex c) { val = c; }           // 保存值
    Add(double r, double i) { val = complex(r, i); }
    void operator()(complex& c) const { c += val; } // 给参数加上那个值
};
```

类Add的对象用一个复数进行初始化，而后通过()调用，它把那个数加到参数上。例如，

```
void h(vector<complex>& aa, list<complex>& ll, complex z)
{
    for_each(aa.begin(), aa.end(), Add(2, 3));
    for_each(ll.begin(), ll.end(), Add(z));
}
```

这就能将`complex(2, 3)`加到向量的每个元素上，并将`z`加到表的每个元素上。注意，`Add(z)`将构造起一个对象，该对象由`for_each()`反复使用。这个对象并不是一个简单的调用一次或者反复调用的函数，被反复调用的函数实际上是`Add(z)`的`operator()()`。

这种方式之所以能工作，还因为`for_each()`是一个模板，它反复地应用其第三个参数的`()`，根本不管这个参数到底是什么东西：

```
template<class Iter, class Fct> Fct for_each(Iter b, Iter e, Fct f)
{
    while (b != e) f(*b++);
    return f;
}
```

初看起来，这种技术似乎有点深奥难解，但实际上它非常简单和高效，而且特别有用（3.8.5节、18.4节）。

`operator()()`的另外一些很流行的应用包括作为子串运算符，以及作为多维数组的下标运算符（22.4.5节）等。

`operator()()`必须作为成员函数。

11.10 间接

间接运算符`->`可以被定义为一个一元的后缀运算符。也就是说，给出了类

```
class Ptr {
    // ...
    X* operator->();
};
```

就可以用类**Ptr**的对象访问类**X**的成员，其方式就像所用的不过是一个指针。例如，

```
void f(Ptr p)
{
    p->m = 7;      // (p.operator -> ()) -> m=7
}
```

从对象**p**到指针**p.operator -> ()**的转换并不依赖于被指向对象的成员**m**，这也就是为什么应该把**operator -> ()**看做一元后缀运算符的理由。但是，无论如何这里并没有引进一种新语法形式，所以，在**->**之后仍然要求写一个成员名字。例如，

```
void g(Ptr p)
{
    X* q1 = p->;      // 语法错
    X* q2 = p.operator->(); // ok
}
```

重载**->**的最有用之处是创建所谓的“灵巧指针”，也就是一种行为像是指针的对象，在通过它们去访问对象时，可以执行一些附加的操作。例如，我们可以定义一个类**Rec_ptr**，用于访问存储在磁盘里的**Rec**类，**Rec_ptr**的构造函数以名字为参数，利用它查找磁盘中的对象，在通过**Rec_ptr**访问时，**Rec_ptr::operator -> ()**将对象搬进主存，而**Rec_ptr**的析构函数最终把更新后的对象写回磁盘：

```
class Rec_ptr {
    const char* identifier;
    Rec* in_core_address;
    // ...
public:
    Rec_ptr(const char* p) : identifier(p), in_core_address(0) { }
    ~Rec_ptr() { write_to_disk(in_core_address, identifier); }
    Rec* operator->();
};

Rec* Rec_ptr::operator->()
{
    if (in_core_address == 0) in_core_address = read_from_disk(identifier);
    return in_core_address;
}
```

Rec_ptr可以按如下方式使用：

```
struct Rec {      // Rec_ptr指向的Rec
    string name;
    // ...
};

void update(const char* s)
{
    Rec_ptr p(s);      // 对s得到一个Rec_ptr

    p->name = "Roscoe";    // 更新s；如果需要，第一次将从磁盘提取
    // ...
}
```

很自然，一个实际的**Rec_ptr**可能是个模板，而**Rec**类型则是它的参数。此外，实际程序可能需要包含错误处理代码，并采用某种不那么简单的方式与磁盘交互。

对于常规指针，**->**的使用和一元*****与**[]**的某些使用方式意义相同。有了

```
Y* p;
```

下面的关系成立：

```
p->m == (*p).m == p[0].m
```

与其他地方一样，用户定义运算符并不保证这些关系。如果需要，也可以提供这些等价性：

```
class Ptr_to_Y {
    Y* p;
public:
    Y* operator->() { return p; }
    Y& operator*() { return *p; }
    Y& operator[](int i) { return p[i]; }
};
```

如果你提供了这些运算符之中的不止一个，提供这些等价关系可能是明智之举，与此类似的是，如果对于某个类提供了++、+=、=和+，那么对于简单变量x，保证++x和x+=1与x=x+1作用相同也是比较明智的。

运算符->的重载对于很大一类很有意思的程序极其重要，这并不仅仅是一种次要的好玩之物。产生这种情况的原因是，间接是一个很重要的概念，而重载的->提供了一种在程序里表示间接的清晰、直接而有效的方式。迭代器（第19章）是这个说法的一个重要佐证。认识运算符->的另一种方式是把它看成C++里提供的一种受限的，但也非常有用的委托机制（24.3.6节）。

运算符->必须是成员函数。如果使用，它的返回类型必须是一个指针，或者是一个你可以应用->运算符的类对象。在声明模板类时，通常都不用operator->()，所以，把对返回类型的约束条件的检查推迟到实际使用时再做也是有意义的。

11.11 增量和减量

一旦人们定义了“灵巧指针”，他们也常常会决定需要提供增量运算符++和减量运算符--，去模拟这些运算符对于内部类型的使用。如果实际目标就是用具有同样语义的“灵巧指针”去取代常规指针类型，只是加上了一些运行时的错误检查，这一做法的意思很明显，也是必要的。例如，考虑一个有些麻烦的传统程序

```
void f1(T a)          // 传统使用
{
    T v[200];
    T* p = &v[0];
    p--;
    *p = a;    // 呜呼：p跑出范围，没有捕捉到
    ++p;
    *p = a;    // ok
}
```

我们可能希望用类Ptr_to_T的对象来取代指针p，这种对象只在实际指向一个对象的时候才能间接访问。我们还应该保证能对这种对象做增量和减量，只要它指向的是一个数组里的对象，且增量减量操作产生数组中的另一个对象。我们可能喜欢这样的代码

```
class Ptr_to_T {
    // ...
```

```

};

void f2(T a)          // 带检查的
{
    T v[200];
    Ptr_to_T p(&v[0], v, 200);
    p--;
    *p = a;    // 运行错误: p越界
    ++p;
    *p = a;    // ok
}

```

增量和减量运算符是C++ 里仅有的能够同时用于前缀和后缀的运算符。因此, 我们也必须为 *Ptr_to_T* 定义前缀和后缀形式的增量与减量。例如,

```

class Ptr_to_T {
    T* p;
    T* array;
    int size;
public:
    Ptr_to_T(T* p, T* v, int s);    // 约束到大小为s的数组v, 初始值为p
    Ptr_to_T(T* p);                // 约束到单个对象, 初始值为p

    Ptr_to_T& operator++();         // 前缀
    Ptr_to_T operator++(int);       // 后缀

    Ptr_to_T& operator--();         // 前缀
    Ptr_to_T operator--(int);       // 后缀

    T& operator*();                // 前缀
};

```

这里用 *int* 参数指明这个函数是为 ++ 的后缀使用而调用的。这个 *int* 绝不会用, 该参数根本就是虚设的, 只是为了区别前缀和后缀应用。记住哪个 *operator++* 是前缀有一种方式, 就是说那个没有虚设参数的是前缀, 与其他一元算术运算符和逻辑运算符一样。虚设参数只是用于“奇特的”后缀 ++ 和 -- 运算符。

通过使用 *Ptr_to_T*, 前面的例子等价于

```

void f3(T a)          // 带检查的
{
    T v[200];
    Ptr_to_T p(&v[0], v, 200);
    p.operator--(0);
    p.operator*() = a;    // 运行错误: p越界
    p.operator++();
    p.operator*() = a;    // ok
}

```

完整的类 *Ptr_to_T* 留做练习 (11.14[19])。将它进一步加工为一个模板, 其中用异常来报告运行时错误是另一个练习 (14.12[2])。将运算符 ++ 和 -- 用于迭代的例子可以在 19.3 节看到。在 13.6.3 节给出了一个能对继承机制正确工作的指针模板。

11.12 一个字符串类

这里是类 *String* 的一个更实际的版本。我将它设计为一个最小的能服务于我的需要的字符

串。这个字符串提供了一种值语义、字符读写操作、检查和不检查的访问、流I/O、用字符串文字量作为文字量、相等和拼接运算符。它将字符串表示为C风格的、用0结束的字符数组，并利用引用计数尽可能地减少复制。写一个更好的字符串类并/或提供更多的功能都是很好的练习（11.14[7~12]）。做完这些之后，我们应丢掉自己的所有练习，去使用标准库的字符串（第20章）。

我的这个近乎实际的**String**使用了三个辅助类：**Srep**用于使一个实际表示能够被几个具有同样值的**String**所共享；**Range**用于在出现范围错误时抛出；**Cref**帮助实现下标运算，这个运算需要区别对待读操作和写操作：

```
class String {
    struct Srep;           // 表示
    Srep *rep;
public:
    class Cref;           // 引用char
    class Range { };      // 用于异常
    // ...
};
```

和其他成员一样，成员类（常常被称做嵌套类）也可以先在类里声明，而后再定义之：

```
struct String::Srep {
    char* s;           // 到元素的指针
    int sz;           // 字符个数
    int n;           // 引用计数

    Srep(int nsz, const char* p)
    {
        n = 1;
        sz = nsz;
        s = new char[sz+1]; // 为结束符增加空间
        strcpy(s, p);
    }

    ~Srep() { delete[] s; }

    Srep* get_own_copy() // 需要时克隆
    {
        if (n==1) return this;
        n--;
        return new Srep(sz, s);
    }

    void assign(int nsz, const char* p)
    {
        if (sz != nsz) {
            delete[] s;
            sz = nsz;
            s = new char[sz+1];
        }
        strcpy(s, p);
    }
private:
    // 防止复制:
    Srep(const Srep&);
    Srep& operator=(const Srep&);
};
```

类**String**提供很普通的一组构造函数、析构函数和赋值运算

```
class String {
    // ...

    String();           // x = ""
    String(const char*); // x = "abc"
    String(const String&); // x = other_string
    String& operator=(const char *);
    String& operator=(const String&);
    ~String();

    // ...
};
```

这个**String**采用值语义。也就是说，在赋值 $s1 = s2$ 之后，两个串 $s1$ 和 $s2$ 是完全分离的，此后对一个**String**的改变对另一个完全没有影响。另一种可能方式是让**String**具有指针语义，这样就可以使得在 $s1 = s2$ 之后对 $s2$ 的修改也能影响 $s1$ 的值。对于有常规算术运算的类型，如复数、向量、矩阵、字符串等，我更喜欢值语义。然而，要使值语义能够负担得起，**String**被实现为一个到实际表示的句柄，而且只在必要时才复制那个表示：

```
String::String()           // 以空串作为默认值
{
    rep = new Srep(0, "");
}

String::String(const String& x) // 复制构造函数
{
    x.rep->n++;
    rep = x.rep;    // 共享表示
}

String::~~String()
{
    if (--rep->n == 0) delete rep;
}

String& String::operator=(const String& x)    // 复制赋值
{
    x.rep->n++;           // 保护，防止“st = st”
    if (--rep->n == 0) delete rep;
    rep = x.rep;         // 共享表示
    return *this;
}
```

伪装的复制运算以**const char*** 作为参数，以提供字符串文字量：

```
String::String(const char* s)
{
    rep = new Srep(strlen(s), s);
}

String& String::operator=(const char* s)
{
    if (rep->n == 1)           // 再利用Srep
        rep->assign(strlen(s), s);
    else {                     // 使用新Srep
        rep->n--;
    }
}
```

```

        rep = new Srep(strlen(s), s);
    }
    return *this;
}

```

对字符串的访问运算符的设计是一个难题，因为理想的访问应能通过很方便的形式（即采用下标[]），提供最高的效率，而且要有范围检查。不幸的是，你将无法同时拥有这些性质。我在这里的选择是：用稍微有一点不方便的形式提供高效的不检查的操作；再加上一种具有方便记法形式的效率稍低的带检查的操作：

```

class String {
    // ...

    void check(int i) const { if (i<0 || rep->sz<=i) throw Range(); }

    char read(int i) const { return rep->s[i]; }
    void write(int i, char c) { rep=rep->get_own_copy(); rep->s[i]=c; }

    Cref operator[] (int i) { check(i); return Cref(*this, i); }
    char operator[] (int i) const { check(i); return rep->s[i]; }

    int size() const { return rep->sz; }

    // ...
};

```

这里的想法是用[]为常规使用提供带检查的访问，但也允许用户做优化，对一系列访问只做一次检查。例如，

```

int hash(const String& s)
{
    int h = s.read(0);
    const int max = s.size();
    for (int i = 1; i<max; i++) h ^= s.read(i)>>1; // 不加检查地访问s
    return h;
}

```

要定义一个像[]那样同时用于读和写的运算符是比较困难的工作，除非我们能接受如下的方式：简单地返回一个引用之后让用户去决定随便怎么使用它。在这里，这种方式就很不合理，因为我已定义了String，其中通过赋值、传递值参数等都能导致String之间共享表示，直到某处实际地向一个String写入，这时，也只有到这时才做表示的复制。这种技术通常称做写时复制。实际的复制由String::Srep::get_own_copy()完成。

为使这些访问函数能在线化，它们的定义就必须放在Srep的定义位于作用域里的地方。这就意味着或者将Srep定义在String里面，或者把访问函数放在String之外，定义为inline，放在Srep的定义之后（11.14[2]）。

为了区分读和写，在采用非const对象调用时，String::operator[]()返回一个Cref。Cref在行为上就像是char&，除了向其中写时它将会调用String::Srep::get_own_copy()：

```

class String::Cref {          // 引用s[i]
friend class String;
    String& s;
    int i;
    Cref(String& ss, int ii) : s(ss), i(ii) {}
public:

```

```

    operator char() const { return s.read(i); } // 产生值
    void operator=(char c) { s.write(i, c); } // 修改值
};

```

例如,

```

void f(String s, const String& r)
{
    char c1 = s[1]; // c1 = s.operator[](1).operator char()
    s[1] = 'c'; // s.operator[](1).operator = ('c')

    char c2 = r[1]; // c2 = r.operator[](1)
    r[1] = 'd'; // 错误: 给const赋值, r.operator[](1) = 'd'
}

```

请注意, 对于非const对象s, s.operator[](I) 也就是Cref(s, I)。

为了完成类String, 我还提供了一组有用的函数:

```

class String {
    // ...

    String& operator+=(const String&);
    String& operator+=(const char*);

    friend ostream& operator<<(ostream&, const String&);
    friend istream& operator>>(istream&, String&);

    friend bool operator==(const String& x, const char* s)
        { return strcmp(x.rep->s, s) == 0; }

    friend bool operator==(const String& x, const String& y)
        { return strcmp(x.rep->s, y.rep->s) == 0; }

    friend bool operator!=(const String& x, const char* s)
        { return strcmp(x.rep->s, s) != 0; }

    friend bool operator!=(const String& x, const String& y)
        { return strcmp(x.rep->s, y.rep->s) != 0; }

};

String operator+(const String&, const String&);
String operator+(const String&, const char*);

```

为节省篇幅, 我把I/O和拼接操作都留做练习。

主程序就是简单地试验String的运算符

```

String f(String a, String b)
{
    a[2] = 'x';
    char c = b[3];
    cout << "in f: " << a << ' ' << b << ' ' << c << '\n';
    return b;
}

int main()
{
    String x, y;
    cout << "Please enter two strings\n";
    cin >> x >> y;
    cout << "input: " << x << ' ' << y << '\n';
    String z = x;
}

```

```

    y = f(x, y);
    if (x != z) cout << "x corrupted!\n";
    x[0] = '!';
    if (x == z) cout << "write failed!\n";
    cout << "exit: " << x << ' ' << y << ' ' << z << '\n';
}

```

这个String可能缺少许多你认为重要的甚至是必不可少的特征。例如，它没有提供操作来产生其值的C字符串表示（11.14[10]，第20章）。

11.13 忠告

- [1] 定义运算符主要是为了模仿习惯使用方式；11.1节。
- [2] 对于大型运算对象，请使用const引用参数类型；11.6节。
- [3] 对于大型的结果，请考虑优化返回方式；11.6节。
- [4] 如果默认复制操作对一个类很合适，最好是直接用它；11.3.4节。
- [5] 如果默认复制操作对一个类不合适，重新定义它，或者禁止它；11.2.2节。
- [6] 对于需要访问表示的操作，优先考虑作为成员函数而不是作为非成员函数；11.5.2节。
- [7] 对于不访问表示的操作，优先考虑作为非成员函数而不是作为成员函数；11.5.2节。
- [8] 用名字空间将协助函数与“它们的”类关联起来；11.2.4节。
- [9] 对于对称的运算符采用非成员函数；11.3.2节。
- [10] 用()作为多维数组的下标；11.9节。
- [11] 将只有一个“大小参数”的构造函数做成explicit；11.7.1节。
- [12] 对于非特殊的使用，最好是用标准string（第20章）而不是你自己的练习；11.12节。
- [13] 要注意引进隐式转换的问题；11.4节。
- [14] 用成员函数表达那些需要左值作为其左运算对象的运算符；11.3.5节。

11.14 练习

1. (*2) 在下面程序中，各个表达式里使用了哪些转换？

```

struct X {
    int i;
    X(int);
    X operator+(int);
};

struct Y {
    int i;
    Y(X);
    Y operator+(X);
    operator int();
};

extern X operator*(X, Y);
extern int f(X);

X x = 1;
Y y = x;
int i = 2;

int main()

```

```
{
    i + 10;    y + 10;    y + 10 * y;
    x + y + i; x * x + i;  f(7);
    f(y);     y + y;     106 + y;
}
```

修改这个程序使之能够运行并打印出每个合法表达式的值。

2. (*2) 完成并测试11.12节的*String*类。
3. (*2) 定义类*INT*并使之在行为上完全像*int*。提示：定义*INT::operator int()*。
4. (*1) 定义类*RINT*并使之在行为上像*int*，除了所允许运算符只有 +（一元和二元）、-（一元和二元）、*、/、%。提示：不要定义*RINT::operator int()*。
5. (*3) 定义类*LINT*并使之在行为上完全像*RINT*，除了它使用至少64位表示之外。
6. (*4) 定义一个实现任意精度算术的类。通过求1 000的阶乘来测试它。提示：你将需要采用类似*String*类的方式管理内存。
7. (*2) 为*String*定义一个外在的迭代器

```
class String_iter {
    // 引用串和串中的元素
public:
    String_iter(String& s);        // 对s的迭代器
    char& next();                  // 引用下一个元素

    // 你选择的更多操作
};
```

从实用、程序设计风格和效率诸方面将它与为*String*提供一个内部迭代器的方式做比较（迭代器是一个概念，它提供*String*的当前元素以及与该元素有关的操作）。

8. (*1.5) 通过重载()为*String*类提供一个子串运算符。你还希望对一个串执行哪些操作？
9. (*3) 设计*String*类，使子串操作可以用在赋值的左边。首先写出一个版本，它允许给串赋一个同样长度的子串；而后再写一个版本允许长度变化。
10. (*2) 为*String*定义一个操作，产生其值的C字符串表示。讨论将这个操作作为转换运算符时可能的支持意见和反对意见，讨论为这个C字符串表示分配存储的各种方式。
11. (*2.5) 为*String*类定义和实现一个简单的正则表达式模式匹配功能函数。
12. (*1.5) 修改取自11.14[11]的模式匹配功能函数，使之能对标准库*string*使用。请注意，你不能修改*string*的定义。
13. (*2) 写一个程序，它通过运算符重载和宏等去表现其难读性。一些想法：对于*INT*定义 + 和 -，将它们的意思颠倒过来；而后再用宏将*int*定义为*INT*。采用引用类型参数重新定义各种常见函数。写一些令人误解的注释也可以造成许多混乱。
14. (*3) 与朋友交换11.14[13]的结果。不去运行它，而是设法弄清你朋友的程序做了些什么。当你完成了这个练习后，你或许知道应该避免一些什么了。
15. (*2) 将*Vec4*定义为4个*float*的向量类型。为*Vec4*定义*operator[]*，为向量和浮点数的组合定义*operator+*、*-*、***、*/*、*=*、*+=*、*-=*、**=*和*/=*。
16. (*3) 定义类*Mat4*为4个*Vec4*的向量，为*Mat4*定义*operator[]*来返回*Vet4*。为这个类型定义常见的矩阵运算。为*Mat4*定义一个做高斯消去法的函数。
17. (*2) 定义类*Vector*，它类似于*Vec4*，但是其大小通过构造函数*Vector::Vector(int)*的参数给出。

18. (*3) 定义类 *Matrix*，它类似于 *Mat4*，但是其各个维通过构造函数 *Matrix::Matrix(int, int)* 的参数给出。
19. (*2) 完成11.11节的类 *Ptr_to_T* 并测试之。为完全起见，*Ptr_to_T* 至少必须定义好运算符 *+*、*->*、*=*、*++* 和 *--*。在实际地间接访问某个野指针之前不要产生运行时错误。
20. (*1) 给定两个结构

```
struct S { int x, y; };  
struct T { char* p; char* q; };
```

写一个类 *C*，使它可以使使用某 *S* 和 *T* 中的 *x* 和 *p*，就像 *x* 和 *p* 原本就是 *C* 的成员似的。

21. (*1.5) 定义类 *Index* 保存指数函数 *mypow(double, Index)* 的指数。找出一种方法让 *2**I* 能够调用 *mypow(2, I)*。
22. (*2) 定义类 *Imaginary* 表示虚数，基于它定义类 *Complex*。实现各种基本算术运算。

第12章 派 生 类

没必要时不重复任何东西。

——W. Occam

概念和类——派生类——成员函数——构造和析构——类层次结构——类型域——虚函数——抽象类——传统的类层次结构——抽象类作为界面——对象创建的局部化——抽象类和类层次结构——忠告——练习

12.1 引言

C++从Simula那里借来了以类作为用户定义类型的概念，以及类层次结构的概念。此外，它还借来了有关系统设计的思想：类应该用于模拟程序员的和应用的世界里的概念。C++ 提供了一些语言结构以直接支持这些设计概念。但在另一方面，使用支持设计概念的语言特征与最有效地使用C++ 还是有所不同，如果只使用这些语言结构作为对更传统程序设计的类型的一种表示形式，那就更是丢掉了C++ 最关键最强有力的东西。

一个概念不会孤立地存在，它总与一些相关的概念共存，并在与相关概念的相互关系中表现出它的大部分力量。举个例子，请试一试去解释什么是汽车，很快你就会引出许多概念：车轮、引擎、司机、人行横道线、卡车、救护车、公路、汽油、超速罚单、汽车旅店等。因为我们要用类表示概念，问题就变成了如何去表示概念之间的关系。然而，我们无法直接在程序语言里表述任意的关系。即使能这样做，我们也未必想去做它。我们的类应该定义的比日常概念更窄一些——而且也更精确。派生类的概念及其相关的语言机制使我们能表述一种层次性的关系，也就是说，表述一些类之间的共性。例如，圆和三角形的概念之间有关系，因为它们都是形状，即它们共有着形状这个概念。因此，我们就必须明确地定义类***Circle***和类***Triangle***，使之共有类***Shape***。在程序里表示出一个圆和一个三角形，然而却没有涉及到形状的概念，就应该认为是丢掉了某些最基本的东西。本章就是对这个简单思想的内涵的一个探索，这个思想就是通常称为面向对象的程序设计的基础。

这里对语言特征和技术的展示仍将从简单而具体的事物开始，逐步进展到更复杂更抽象的事物。对于大部分程序员而言，这也是从熟悉之物到更加未知的世界的一个旅程。这并不是从“糟糕的老技术”到“惟一的正确途径”的简单过渡。当我指出某种技术的局限性，作为通往另一种技术的推动力时，我总是在一个特定问题的环境中做这件事情；而对于不同的问题或者其他环境，第一种技术很可能反而成为更好的选择。人们已经用这里展示的所有技术构造出许许多多实用软件。这里的目标就是帮助你获得对这些技术的充分理解，以便在面对实际问题时，能在它们之中做出明智的有条不紊的选择。

在这一章里，我要首先介绍支持面向对象程序设计的基本语言特征；而后将在一个大例子的环境中，讨论如何使用这些特征去开发结构良好的程序。支持面向对象程序设计的其他

特征，例如多重继承和运行时类型识别，将在第15章讨论。

12.2 派生类

现在来考虑做一个程序，处理某公司所雇佣人员的问题。这个程序可能包含如下的一种数据结构：

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};
```

下一步，我们可能需要去定义经理：

```
struct Manager {
    Employee emp;           // 经理的雇佣记录
    list<Employee*> group;   // 所管理的人员
    short level;
    // ...
};
```

一个经理同时也是一个雇员，所以在`Manager`对象的`emp`成员里存储着`Employee`数据。这对于读程序的人而言是很明显的——特别是细心的读者，但是却没有给编译器或者其他工具提供有关`Manager`也是`Employee`的任何信息。一个`Manager*`就不是`Employee*`，所以，在要求一个的地方也就无法简单地使用另一个。特别是，如果不写出一些特殊代码，你将无法把一个`Manager`放进一个`Employee`的表里。我们当然可以做这件事，或者是对`Manager*`做显式的类型转换，或者是将其`emp`成员的地址存入`Employee`的表。但是，这两种解决方案都不优美，也都是相当不清晰的。正确的途径应该能够把`Manager`也是`Employee`的事实明确地表述出来，再加上少量的信息：

```
struct Manager : public Employee {
    list<Employee*> group;
    short level;
    // ...
};
```

这个`Manager`是由`Employee`派生的，反过来说就是，`Employee`是`Manager`的一个基类。类`Manager`包含了类`Employee`的所有成员（`first_name`，`department`等），再加上它自己的一些成员（`group`，`level`等）。

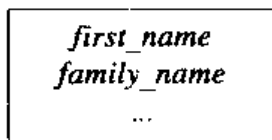
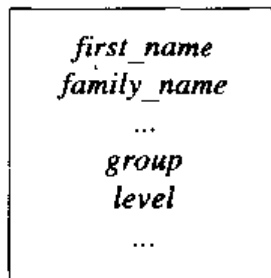
派生关系常用图形表示，画出从派生类到基类的一个箭头，指明派生类引用了它的基类（而不是相反）：



派生类常常被说成是从它的基类继承了各种性质，因此这个关系也被称为继承。基类有时被

称做超类，派生类被称做子类。然而这一对术语却常常把人们搞糊涂，因为他们看到派生类对象的数据是基类对象的数据的一个超集。派生类通常都比基类更大，即是说它保存了更多数据，提供了更多的函数。

对于派生类概念的一种常见且有效的实现方式，就是将派生类的对象也表示为一个基类的对象，只是将那些特别属于派生类的信息附加在最后。例如，

Employee:*Manager:*

按照这种方式从**Employee**派生出**Manager**，就使**Manager**成为**Employee**的一个子类型，使**Manager**可以用在能够接受**Employee**的任何地方。例如，我们现在就可以建立起一个**Employee**的表，而其中的一些元素是**Manager**：

```

void f(Manager m1, Employee e1)
{
    list<Employee*> elist;
    elist.push_front(&m1);
    elist.push_front(&e1);
    // ...
}

```

因为**Manager**（也）是**Employee**，所以**Manager*** 就可以当作**Employee*** 使用。然而，因为**Employee**不一定是**Manager**，所以**Employee*** 就不能当做**Manager*** 用。总而言之，如果类**Derived**有一个公用基类（15.3节）**Base**，那么就可以用**Derived*** 给**Base*** 类型的变量赋值，不需要显式的类型转换。而相反的方向，从**Base*** 到**Derived*** 则必须显式转换。例如，

```

void g(Manager mm, Employee ee)
{
    Employee* pe = &mm;    // 可以: Manager都是Employee
    Manager* pm = &ee;     // 错误: Employee不一定是Manager
    pm->level = 2;          // 灾难: ee没有level

    pm = static_cast<Manager*>(pe);    // 蛮力: 这个可以, 因为pe
                                      // 指向的是Manager mm
    pm->level = 2;          // 没问题: pe指向的是Manager mm, 'level'
}

```

换句话说，在通过指针或者引用方式操作时，派生类的对象可以当做基类的对象看待和处理。反过来则不真。关于**static_cast**和**dynamic_cast**的使用将在第15.4.2节讨论。

用一个类作为基类，相当于声明一个该类的（匿名）对象。所以，要想作为基类，这一个类就必须有定义（5.7节）：

```

class Employee;    // 只声明，不定义
class Manager : public Employee { // 错误: Employee无定义

```

```

    // ...
};

```

12.2.1 成员函数

在实际中，像*Employee*和*Manager*这样的简单数据结构不太令人感兴趣，它们也不是特别有用。我们需要给出更多的信息，将它们做成真正的类型，提供适当的表示有关概念的一组函数，而且，在这样做时又应该避免使我们依赖于特殊的表示细节。例如，

```

class Employee {
    string first_name, family_name;
    char middle_initial;
    // ...
public:
    void print() const;
    string full_name() const
        { return first_name + ' ' + middle_initial + ' ' + family_name; }
    // ...
};

class Manager : public Employee {
    // ...
public:
    void print() const;
    // ...
};

```

派生类的成员可以使用其基类的公用的——和保护的（见15.3节）——成员，就像它们也是在基类里声明的一样。例如，

```

void Manager::print() const
{
    cout << "name is " << full_name() << '\n';
    // ...
}

```

但是，派生类不能使用基类的私用名字：

```

void Manager::print() const
{
    cout << " name is " << family_name << '\n';    // 错误!
    // ...
}

```

Manager::print() 的第二个版本将无法编译。派生类的成员也没有访问其基类的私用成员的特许权，所以*family_name*对于*Manager::print()*而言是不可访问的。

这一点开始可能会使一些人吃惊。那么请考虑另一种安排：派生类的成员函数能够访问其基类的私用成员。这样就会使私用成员这个概念退化成为一种毫无意义的东西，因为程序员可以简单地通过从一个类出发的派生而获得对其私用成员的访问权。进一步说，我们也不能再通过查看声明为某个类的所有成员和友元的函数，而确定对该类私用成员的所有使用。人们将必须去检查整个程序的每个源文件，去找出所有的派生类，而后检查这些类里的每一个函数；还要找出由这些类派生出的每一个类，如此等等。这种处理方式，按最好的情况说，

也是令人讨厌的，常常是不实际的。在能够接受这些的地方，可以采用`protected`成员——而不是`private`成员。对于派生类的成员而言，保护成员就像是公用成员；但对于其他函数它们则像是私用成员（15.3节）。

一般来说，最清晰的设计是派生类只使用它的基类的公用成员。例如，

```
void Manager::print() const
{
    Employee::print(); // 打印Employee信息
    cout << level;      // 打印Manager的特殊信息
    // ...
}
```

注意，在这里必须用`::`，因为在`Manager`里重新定义了`print()`。名字的这种重新使用是很典型的。不当心的人可能这样写

```
void Manager::print() const
{
    print(); // 糟糕！
    // 打印Manager的特殊信息
}
```

然后就会发现这个程序被牵涉进一个未预料到的无穷递归里。

12.2.2 构造函数和析构函数

有些派生类需要构造函数。如果某个基类中有构造函数，那么就必须调用这些构造函数中的某一个。默认构造函数可以被隐含地调用，但是，如果一个基类的所有构造函数都有参数，那么就必须显式地调用其中的某一个。考虑

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& n, int d);
    // ...
};

class Manager : public Employee {
    list<Employee*> group; // 所管理的人员
    short level;
    // ...
public:
    Manager(const string& n, int d, int lvl);
    // ...
};
```

基类构造函数的参数应在派生类构造函数的定义中有明确描述。在这方面，基类的行为恰恰就像是派生类的一个成员（10.4.6节）。例如，

```
Employee::Employee(const string& n, int d)
    : family_name(n), department(d) // 初始化成员
{
    // ...
}
```

```

}

Manager::Manager(const string& n, int d, int lvl)
: Employee(n, d),          // 初始化基类
  level(lvl)               // 初始化成员
{
    // ...
}

```

派生类的构造函数只能描述它自己的成员和自己的直接基类的初始式，它不能直接去初始化基类的成员。例如，

```

Manager::Manager(const string& n, int d, int lvl)
: family_name(n),          // 错误：在Manager里没有family_name的声明
  department(d),          // 错误：在Manager里没有department的声明
  level(lvl)
{
    // ...
}

```

这个定义中包含了三个错误：它没有调用**Employee**的构造函数，而且还两次企图去直接初始化**Employee**的成员。

类对象的构造是自下而上进行的：首先是基类，而后是成员，再后才是派生类本身。类对象的销毁则正好以相反的顺序进行：首先是派生类本身，而后是成员，再后才是基类。成员和基类的构造严格按照在类声明中的顺序，它们的销毁则按照相反的顺序进行。另见10.4.6节和15.2.4.1节。

12.2.3 复制

类对象的复制由复制构造函数和赋值操作定义（10.4.4.1节）。考虑

```

class Employee {
    // ...
    Employee& operator=(const Employee&);
    Employee(const Employee&);
};

void f(const Manager& m)
{
    Employee e = m;          // 从m的Employee部分创建e
    e = m;                   // 用m的Employee部分给e赋值
}

```

由于**Employee**的复制函数根本不知道**Manager**的任何情况，所以只有**Manager**的**Employee**部分被复制。这种情况通常被称为**切割**，它可能成为使人诧异和产生错误的根源。需要在类层次结构中传递类对象的指针和引用，其中的一个原因就是为了避免切割问题。另外的原因是为了维持多态性行为（2.5.4节、12.2.6节）和保证效率。

注意，如果你没有定义复制赋值运算符，编译器就会为你生成一个（11.7节）。这也意味着赋值运算符是不继承的。构造函数也是绝不继承的。

12.2.4 类层次结构

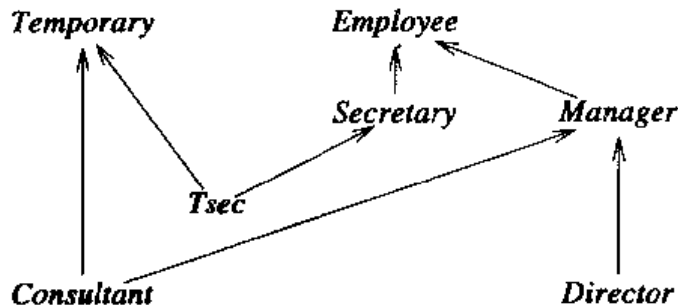
派生类本身也可以作为基类。例如，

```
class Employee { /* ... */ };
class Manager : public Employee { /* ... */ };
class Director : public Manager { /* ... */ };
```

这样，一组相关的类按照习惯被称做一个类层次结构。这种层次结构的最常见形式是一棵树，当然它也可能具有更一般的图结构。例如，

```
class Temporary { /* ... */ };
class Secretary : public Employee { /* ... */ };
class Tsec : public Temporary, public Secretary { /* ... */ };
class Consultant : public Temporary, public Manager { /* ... */ };
```

或画成图



可以看出，C++能够表示类的有向无环图，这方面的情况将在15.2节解释。

12.2.5 类型域

为了使派生类不仅仅是一种完成声明的方便简写形式，我们就必须解决下面的问题：对给定的一个类型为 *Base** 的指针，被指的对象到底属于哪个派生类型呢？这个问题有四种基本的解决方案：

- [1] 保证被指的只能是惟一类型的对象（2.7节，第13章）。
- [2] 在基类里安排一个类型域，供函数检查。
- [3] 使用 *dynamic_cast*（15.4.2节、15.4.5节）。
- [4] 使用虚函数（2.5.5节、12.2.6节）。

指向基类的指针常常被用于设计各种容器类，如集合、向量和表等。这时，解决方案1将产生出同质的表，即所有元素都具有同样类型的表。解决方案2、3、4可用于构造出异质的表，即一些不同类型的对象（的指针）的表。解决方案3是解决方案2的由语言支持的变形。解决方案4是解决方案2的一种特殊的类型安全的变形。解决方案1和解决方案4的组合特别有意思，而且威力强大，它们能产生出比解决方案2和解决方案3更清晰的代码。

让我们首先考察简单的类型域解决方案，看看为什么应该极力避免它。经理/雇员的例子可以重新定义为如下形式：

```
struct Employee {
    enum Empl_type { M, E };
    Empl_type type;

    Employee() : type(E) {}

    string first_name, family_name;
    char middle_initial;
```

```

    Date hiring_date;
    short department;
    // ...
};

struct Manager : public Employee {
    Manager() { type = M; }

    list<Employee*> group;    // 被管理的人员
    short level;
    // ...
};

```

有了这个定义，我们就可以写出一个函数，打印与每个`Employee`有关的信息

```

void print_employee(const Employee* e)
{
    switch (e->type) {
    case Employee::E:
        cout << e->family_name << '\t' << e->department << '\n';
        // ...
        break;
    case Employee::M:
        {
            cout << e->family_name << '\t' << e->department << '\n';
            // ...
            const Manager* p = static_cast<const Manager*>(e);
            cout << " level " << p->level << '\n';
            // ...
            break;
        }
    }
}

```

并将它用于打印`Employee`的表，如下所示：

```

void print_list(const list<Employee*>& elist)
{
    for (list<Employee*>::const_iterator p = elist.begin(); p != elist.end(); ++p)
        print_employee(*p);
}

```

这确实能行，特别是在由一个人维护的小程序里。然而，这种方法也有一个致命的弱点：它需要依靠程序员按照确定的方式去操纵这些类型，而所用的方式是编译器无法检查的。由于像`print_employee()`这类函数的组织方式可能利用所涉及的类之间的共同性（很常见），问题通常会变得更糟：

```

void print_employee(const Employee* e)
{
    cout << e->family_name << '\t' << e->department << '\n';
    // ...
    if (e->type == Employee::M) {
        const Manager* p = static_cast<const Manager*>(e);
        cout << " level " << p->level << '\n';
        // ...
    }
}

```

在处理着许多派生类的很大的函数里，要找出隐藏其中的所有这种对类型域的检测将会很困

难。即使能找到它们，要理解其中究竟是怎么回事也很困难。进一步说，增加任何一种新的 *Employee* 都将涉及到修改系统中所有的这种关键性函数——即所有的包含了对类型域进行检测的函数。在每次修改之后，程序员必须去考虑每一个可能需要检测类型域的函数，这实际上意味着需要查看所有关键性的源代码，继而又将导致测试受影响代码的必要开销。使用显式转换也是一个很强的提示：这里可能需要改造。

换句话说，使用类型域是一种极容易出错的技术，它还引起维护中的大麻烦。随着程序规模的增长，这个问题将会变得更加严重，因为类型域的使用实际上是对模块化和数据隐藏理想的亵渎。每个使用类型域的函数都必须知道由这个包含着类型域类派生出的每个类的内部表示和其他的实现细节。

还可以看到，存在着每个导出类都能访问的任何公共数据（例如一个类型域），将刺激人们加入更多的这类数据。于是，基类将变成各种各样的“有用信息”的展示台。这反过来又使基类和派生类的实现以某种我们非常不希望的方式互相纠缠。而为了清晰的设计和简化维护事宜，我们则希望能让分离的东西互相分离，尽可能地避免相互依赖性。

12.2.6 虚函数

虚函数能克服类型域解决方案中的缺陷，它使程序员可以在基类里声明一些能够在各个派生类里重新定义的函数。编译器和装载程序能保证对象和应用于它们的函数之间的正确对应关系。例如，

```
class Employee {
    string first_name, family_name;
    short department;
    // ...
public:
    Employee(const string& name, int dept);
    virtual void print() const;
    // ...
};
```

关键字 *virtual* 指明 *print()* 的作用就像是一个界面，既要服务于本类中定义的 *print()* 函数，也服务于由它派生出的类里所定义的 *print()* 函数。对于派生类里定义了这样的 *print()* 的那些地方，编译器将保证，在每种情况下，对于给定的 *Employee* 对象^①，都一定能调用到正确的 *print()* 函数。

为了使虚函数声明能起到作为在派生类里定义的函数的界面的作用，在派生类里，对有关函数所描述的参数类型就不能与基类中声明的参数类型有任何差异，只有在返回类型上允许小小的改变（15.6.2节）。虚成员函数有时也被称为方法。

在某个虚函数的第一个声明所在的那个类里，该虚函数也必须予以定义（除非它被声明为纯虚函数，12.3节）。例如，

```
void Employee::print() const
{
```

① 注意，*Employee* 的派生类的对象也应看做是 *Employee* 对象。这段话涵盖了如下情况：一个对象属于 *Employee* 的某个派生类，而该派生类定义了自己的 *print()*，那么对该对象“正确的”*print()* 就是该派生类里所定义的那个函数。本书常有“类X的所有对象”之类的说法，请读者注意。——译者注


```

        cout << family_name << '\t' << department << '\n';
        // ...
    }

```

即使没有从某个类派生出其他类，也可以使用其中的虚函数；如果某个派生类不需要自己的虚函数版本，那就完全不必提供自己的版本。在声明派生类时，如果需要的话，就可以简单地提供一个合适的函数。例如，

```

class Manager : public Employee {
    list<Employee*> group;
    short level;
    // ...
public:
    Manager(const string& name, int dept, int lvl);
    void print() const;
    // ...
};

void Manager::print() const
{
    Employee::print();
    cout << "\tlevel " << level << '\n';
    // ...
}

```

如果在派生类里存在一个函数，它具有与基类中的某个虚函数同样的名字和同样一组参数类型，我们就说它覆盖了这个虚函数的基类版本。除非明确说明要调用的是虚函数的哪一个版本（例如，在调用`Employee::print()`中），否则，在对一个对象调用虚函数时，被选用的总是那个最适于它的覆盖函数。

现在就不需要全局函数`print_employee()`（12.2.5节）了，因为成员函数`print()`已经取代了它的位置。一个`Employee`的表可以按如下方式打印出来

```

void print_list(const list<Employee*>& s)
{
    for (list<Employee*>::const_iterator p = s.begin(); p != s.end(); ++p)    // 见2.7.2节
        (*p)->print();
}

```

或者甚至是

```

void print_list(const list<Employee*>& s)
{
    for_each(s.begin(), s.end(), mem_fun(&Employee::print));    // 见3.8.5节
}

```

各个`Employee`都将依其类型打印。例如，

```

int main()
{
    Employee e("Brown", 1234);
    Manager m("Smith", 1234, 2);
    list<Employee*> empl;
    empl.push_front(&e);    // 见2.5.4节
    empl.push_front(&m);
    print_list(empl);
}

```

将产生出

```
Smith 1234
    level 2
Brown 1234
```

注意，这样做完全可行，即使`print_list()`是在还没有考虑到特定的派生类`Manager`之前写好并编译好的。这是类机制中最关键的一个方面。只要用得好，它能够成为面向对象程序设计的基石，并能为一个不断演化的程序提供某种程度的稳定性。

从`Employee`的函数中取得“正确的”行为，而又不依赖于实际使用的到底是哪一种`Employee`，这就是所谓的多态性。一个带有虚函数的类型被称为是一个多态类型。要在C++里取得多态性的行为，被调用的函数就必须是虚函数，而对象则必须是通过指针或者引用去操作的。如果直接操作一个对象（而不是通过指针或引用），它的确切类型就已经为编译器所知，因此也就不需要运行时的多态性了。

显而易见，为了能够实现多态性，编译器必须在类`Employee`的每个对象里存储某种类型信息，并在需要调用虚函数`print()`的正确版本时利用这些信息。在典型的实现里，所需空间只是一个指针（2.5.5节）。只有那些包含了虚函数的类的对象才需要这点空间——而不是任何对象，甚至不是任何派生类的对象。你只需要为声明了虚函数的类付出这种额外开销。如果你原本采用的是另一种方式，采用类型域，你也将为类型域的需要而用掉数量差不多的空间。

通过作用域解析运算符`::`去调用函数，就像在`Manager::print()`里面所做的那样，就能保证不使用虚函数机制。如果不那样写，`Manager::print()`就会陷入无穷递归。使用带限定词的名字还有另一个我们希望的作用，那就是，如果某个`virtual`函数也是`inline`的（这也很常见），那么就可以对用`::`特殊说明的调用使用在线替换。这就使程序员能有一种很有效的方法去处理一些重要的特殊情况，其中某个虚函数针对同一个对象调用了另一个虚函数。`Manager::print()`函数就是这样的一个例子。因为在`Manager::print()`的调用中对象的类型已经确定，在随后调用`Employee::Print()`时，就不必再次去动态确定它了。

很值得指出，对于虚函数调用的传统而又明显的实现方式就是一个简单的间接函数调用（2.5.5节），所以，对效率的关心不应该使任何人对虚函数的使用犹豫不决，只要在某些地方常规函数调用被认为是足够有效的。

12.3 抽象类

有许多类与`Employee`类似，它们本身很有用，又能作为派生类的基类。对于这样的类而言，前一节所描述的技术也就足够了。但是，并不是所有的类都具有这种模式。有些类，例如类`Shape`，表达的就是一种根本不存在对象的抽象概念。`Shape`的意义只在于它能作为由它所派生出的类的基类，这一点可以从下面事实中看出来：我们根本无法给它的虚函数提供有意义的定义

```
class Shape {
public:
    virtual void rotate(int) { error("Shape::rotate"); } // 不优美
    virtual void draw() { error("Shape::draw"); }
    // ...
};
```

想做出这种没有特定意思的`Shape`虽然很愚蠢，但却是合法的：

```
Shape s; // 愚蠢：“无形的形状”
```

说它愚蠢，是因为对它的每个操作都将导致一个错误。

另一种更好的方式是把类`Shape`的虚函数声明为纯虚函数。用 `= 0` 作为初始式就使虚函数成为“纯虚的”：

```
class Shape {           // 抽象类
public:
    virtual void rotate(int) = 0;    // 纯虚函数
    virtual void draw() = 0;         // 纯虚函数
    virtual bool is_closed() = 0;    // 纯虚函数
    // ...
};
```

如果一个类里存在一个或者几个纯虚函数，这个类就是个抽象类。不能创建抽象类的对象：

```
Shape s; // 错误：抽象类Shape的变量
```

抽象类只能用做界面，作为其他类的基类。例如，

```
class Point { /* ... */ };

class Circle : public Shape {
public:
    void rotate(int) { }           // 覆盖Shape::rotate
    void draw();                   // 覆盖Shape::draw
    bool is_closed() { return true; } // 覆盖Shape::is_closed

    Circle(Point p, int r);

private:
    Point center;
    int radius;
};
```

一个未在派生类里定义的纯虚函数仍旧还是一个纯虚函数，这种情况也将使该派生类仍为一个抽象类。这就使我们可以分步骤地构筑起一个实现：

```
class Polygon : public Shape {           // 抽象类
public:
    bool is_closed() { return true; }    // 覆盖Shape::is_closed
    // ..... draw和rotate尚未覆盖.....
};

Polygon b;    // 错误：声明的是抽象类Polygon对象

class Irregular_polygon : public Polygon {
    list<Point> lp;
public:
    void draw();           // 覆盖Shape::draw
    void rotate(int);       // 覆盖Shape::rotate
    // ...
};

Irregular_polygon poly(some_points);    // 可以（假定有合适的构造函数）
```

抽象类的最重要用途就是提供一个界面，而又不暴露任何实现的细节。例如，一个操作系统可以将其设备驱动程序细节都隐藏到一个抽象类的后面

```

class Character_device {
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...) = 0;
    virtual ~Character_device() { }    // 虚析构函数
};

```

我们随后就可以将驱动程序描述为由**Character_device**派生的类，并经由这个界面去操作各种各样的驱动程序。虚析构函数的重要性将在12.4.2节给出解释。

引进了抽象类的概念之后，我们就有了以类作为基本构件，按照模块化的方式写出完整程序的最基本的设施了。

12.4 类层次结构的设计

现在考虑一个简单的设计问题：为某个程序提供一种方式，通过它可以从某种用户界面取得一个整数。做这件事的方式真是无穷无尽。为了将我们的程序隔离于这些纷繁的变化之外，而又能有机会去利用各种可能的的设计选择，让我们从为这个简单操作定义自己的程序模型入手。我们将把如何使用实际用户界面系统的实现细节留在后面处理。

这里的想法是采用一个类**Ival_box**，它知道能够接受的输入值的取值范围。程序可以向**Ival_box**要求一个值，在必要时要求它去提示用户输入。此外，程序还可以询问**Ival_box**，看看在程序最后一次查看之后用户是否修改过有关的值。

因为存在着实现这种基本想法的许多不同方式，我们必须假定存在多种不同种类的**Ival_box**，例如滑块、用户可以键入一个数的普通输入盒、拨盘、语音交互装置等。

最具普遍性的途径是构造起一个“虚拟用户界面系统”，供应用系统使用。这个系统提供了现存用户界面系统所提供的各种服务中的一些功能，它可以在变化多端的多种不同系统上实现，从而保证了应用代码的可移植性。自然，也完全可以采用其他方法将一个具体应用与用户界面系统隔离。我之所以选择这种途径，是因为它具有通用性，因为它使我可以阐释各种各样的技术和设计权衡，也因为这些技术正是在“真实的”用户界面系统中所用的东西，而且——最重要的——是因为这些技术的应用范围远远超出了用户界面系统这样一个狭窄的领域。

12.4.1 一个传统的层次结构

我们的第一个解决方案采用的是经常能在Simula、Smalltalk和一些老的C++ 程序中看到的类层次结构。

类**Ival_box**定义了所有**Ival_box**的基本界面，并刻画了一种默认的实现，各种更特殊的**Ival_box**可以用自己的实现版本覆盖这个默认实现。此外，我们还声明了为实现基本概念所需要的数据：

```

class Ival_box {
protected:
    int val;
    int low, high;

```

```

    bool changed; // 由用户通过set_value() 修改
public:
    Ival_box(int ll, int hh) { changed = false; val = low = ll; high = hh; }

    virtual int get_value() { changed = false; return val; }
    virtual void set_value(int i) { changed = true; val = i; } // 为用户
    virtual void reset_value(int i) { changed = false; val = i; } // 为应用
    virtual void prompt() {}
    virtual bool was_changed() const { return changed; }
};

```

这些函数的默认实现都非常卓率，放在这里基本上就是为了阐释所需要的语义。比如说，某个实际的类可能会做有关取值范围的检查。

程序员可能会像下面的样子使用这些“*Ival*类”：

```

void interact(Ival_box* pb)
{
    pb->prompt(); // 提醒用户
    // ...
    int i = pb->get_value();
    if (pb->was_changed()) {
        // 新值；做某些事情
    }
    else {
        // 老值还在；做另一些事情
    }
    // ...
}

void some_fct()
{
    Ival_box* p1 = new Ival_slider(0, 5); // 从Ival_box派生的Ival_slider
    interact(p1);

    Ival_box* p2 = new Ival_dial(1, 12);
    interact(p2);
}

```

大部分应用代码都是像`interact()`这样，采用（通过指针）针对普通*Ival_box*的方式写出。按照这种方式，应用系统完全不需要知道*Ival_box*概念可能有的数量庞大的各种各样的变形。有关特定类的知识被隔离在相对很少的一些需要创建有关对象的函数里。这样就能把用户隔离在派生类的实现变化之外。大部分代码可以完全不理睬这样的事实：确实存在着不同种类的*Ival_box*。

为了简化讨论，我不打算去考虑与程序如何等待与输入有关的问题。可能这个程序就是在`get_value()`里等着用户；程序也可能将*Ival_box*关联于某个事件，并准备去响应一个回调；或者程序也可能为*Ival_box*生成一个线程，以后再去查询该线程的状态。在设计用户界面系统时，这些决策都是至关重要的。然而，要在这里讨论它们的任何实际细节，都将偏离有关程序设计技术和语言功能的讨论。这里所描述的设计技术和支持这些技术的语言功能并不是专门针对用户界面的，它们适用于范围远比这些更广泛的许多问题。

不同种类的*Ival_box*被定义成*Ival_box*的派生类。例如，

```

class Ival_slider : public Ival_box {
    // 定义滑块的视觉形式的图形要素等

```

```

public:
    Ival_slider(int, int);

    int get_value();
    void prompt();
};

```

*Ival_box*的数据成员被声明为`protected`，以便从派生类能够访问。这样，*Ival_slider::get_value()*就能把值直接放入*Ival_box::val*里。`protected`成员可以由这个类本身的成员访问，也可以从派生类的成员访问，但一般用户却不能访问（见15.3节）。

除了*Ival_slider*之外，我们还应该定义*Ival_box*概念的其他变形。这可能包括*Ival_dial*，它使你可以通过转一个旋钮去选择某个值；*Flashing_ival_slider*，在你要求它用`prompt()`给出提示时不停地闪烁；*Popup_ival_slider*，它对`prompt()`的反应是在某个固定的位置显现出来，以使用户不容易忽略它。

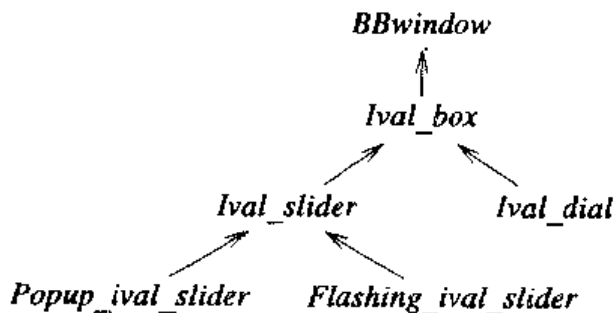
我们从哪里获得这些图形要素呢？大部分图形用户界面系统都提供了一个类，其中定义了屏幕实体的基本性质。所以，如果我们采用来自“Big Bucks Inc.”^①的系统，我们就可以把自己的*Ival_slider*、*Ival_dial*等各个类都做成一种*BBwindow*。达到这种效果的最简单方式就是重写我们的*Ival_box*，让它从*BBwindow*派生。通过这种方式，我们所有的类都继承了*BBwindow*的所有性质。例如，每个*Ival_box*都可以依靠*BBwindow*系统的标准操作集合放在屏幕上，都将遵守图形风格规则，可以改变大小，拖来拖去，如此等等。我们的类层次结构如下

```

class Ival_box : public BBwindow { /* ... */ }; // 应用BBwindow重写
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };

```

其图示是



12.4.1.1 评论

这种设计在许多方面都工作得很好，对于许多问题，这种层次结构都是很好的解决方案。但是，这里也存在着一些麻烦的细节，这些情况促使我们去寻找其他设计途径。

我们后来把*BBwindow*作为*Ival_box*的基类，这样做有些不太对头。使用*BBwindow*并不是*Ival_box*概念的最基本部分，只不过是一个实现细节。从*BBwindow*派生出*Ival_box*，实际

① 作者不愿意提出任何真正的图形用户界面系统，因此在这里随便虚构了一个公司的名字。这也是一种习惯方式。——译者注

上就是把一个实现细节提升到第一级的设计决策里。这种做法也可能是对的，例如，使用“Big Bucks Inc.”所定义的环境可能是我们所在的那个组织经营其业务的一个关键决策。然而，倘若我们还想让自己的*Ival_box*能够实现在“Imperial Bananas”、“Liberated Software”和“Compiler Whizzes”的系统里，事情又会怎么样呢？我们将不得不去维护程序的4个不同版本：

```
class Ival_box : public BBwindow { /* ... */ }; // BB版本
class Ival_box : public CWwindow { /* ... */ }; // CW版本
class Ival_box : public IBwindow { /* ... */ }; // IB版本
class Ival_box : public LSwindow { /* ... */ }; // LS版本
```

存在许多版本必然导致一个版本控制的恶梦。

另一个问题是，每个派生类都共享着在*Ival_box*里声明的基本数据。这个数据当然也是一种实现细节，但它们却潜入了我们的*Ival_box*界面。从实践的角度看，在许多情况下这些数据也是不对的。例如，*Ival_slider*根本不需要特别存储的值，这个值能在用户调用*get_value()*时轻易地由滑块的位置计算出来。一般来说，保存两份相互关联但又不同的数据集合就是自找麻烦，或迟或早会出现某些情况，导致它们失去同步。另外，经验说明，程序员新手很容易以某种不必要的方式迷失在保护数据之中，并由此给程序维护引进一些难题。数据成员最好都是私用的，这将使写派生类的人们无法来搅和它们。更进一步，数据应该放到派生类里，在那里可以根据需要确切地定义它们，又不会把与之无关的派生类的生活搞得更复杂。对于几乎所有情况，保护界面都只应该包含函数、类型和常量^①。

从*BBwindow*派生，得到的利益是使*Ival_box*的用户能直接使用*BBwindow*提供的功能。不幸的是，这也意味着类*BBwindow*的修改将迫使这些用户重新编译，甚至重写他们的代码去应对有关的变化。特别地，大部分C++实现的工作方式都隐含着一种情况：基类对象大小的改变将要求重新编译所有的派生类。

最后，我们的程序也可能需要运行在一个混合式的环境里，其中共存着多个源自不同用户界面系统的窗口。这个情况有可能出现，或者是两种不同的系统以某种方式共用屏幕，或者是我们的程序需要与在另一个不同系统上的用户通信。将自己的用户界面系统“硬连接”到一个系统上，仅仅连到我们作为基类的那个系统，且只以*Ival_box*作为界面，这种方式将无法提供足以应付上述情况所需要的灵活性。

12.4.2 抽象类

好吧！让我们从头再来，构筑起另一个新的类层次结构，解决在对传统层次结构的评论中所提出的那些问题：

- [1] 用户界面系统应该是一个实现细节，对于不希望了解它的用户应是隐蔽的。
- [2] 类*Ival_box*应该不包含数据。
- [3] 在用户界面修改后，不应该要求重新编译那些使用了*Ival_box*这一族类的代码。
- [4] 针对不同界面系统的*Ival_box*应能在我们的程序里共存。

存在着若干不同的途径都可以达到这些目标。我将在这里介绍的是其中的一种能够最清晰地

① 保护界面，指的是一个类中由定义为*protected*访问权限的全体成员构成的界面（更广义的说法是，保护界面也包含了类的公用界面）。这个界面是派生类可以访问，外部不能访问的。——译者注

映射到C++语言中的途径。

首先，我把类*Ival_box*刻画为一个抽象界面

```
class Ival_box {
public:
    virtual int get_value() = 0;
    virtual void set_value(int i) = 0;
    virtual void reset_value(int i) = 0;
    virtual void prompt() = 0;
    virtual bool was_changed() const = 0;
    virtual ~Ival_box() { }
};
```

这比*Ival_box*原来的声明清楚得多。数据没有了，还去掉了那些成员函数的简单实现。丢掉的还有构造函数，因为这里不存在需要去初始化的数据。相反，我增加了一个虚析构函数，以确保将在派生类里定义的数据能被正确地清理。

*Ival_slider*的定义看起来可能是这样

```
class Ival_slider : public Ival_box, protected BBwindow {
public:
    Ival_slider(int, int);
    ~Ival_slider();

    int get_value();
    void set_value(int i);
    // ...
protected:
    // 覆盖BBwindow虚函数的函数
    // 例如, BBwindow::draw(), BBwindow::mouseHit()
private:
    // 滑块所需的数据
};
```

派生类*Ival_slider*从一个抽象类(*Ival_box*)继承，就要求它实现基类中所有的纯虚函数。它同时从*BBwindow*继承来用于做这些事情的功能。因为*Ival_box*为派生类提供的是界面，从它的派生采用的是*public*。因为*BBwindow*只是作为实现辅助，从它的派生采用*protected* (15.3.2节)。这也意味着，使用*Ival_slider*的程序员不能直接使用*BBwindow*所定义的功能。由*Ival_slider*提供的界面就是它从*Ival_box*继承来的，再加上*Ival_slider*里显式声明的。我采用*protected*派生而不是更严格（通常也更安全）的*private*，是为了使由*Ival_slider*派生出的类还能使用*BBwindow*。

从多于一个类出发的派生方式通常被称为多重继承 (15.2节)。请注意，*Ival_slider*必须覆盖来自*Ival_box*和*BBwindow*两者的函数，因此它必须直接或间接地从这两者派生。如12.4.1.1节所示，通过将*BBwindow*作为*Ival_box*的基类，使*Ival_slider*间接地由*BBwindow*派生出来也可行，但那样做会带来我们所不希望的副作用。与此类似，将“实现类”*BBwindow*作为*Ival_box*的一个成员也不能解决问题，因为一个类没办法覆盖其成员的虚函数 (24.3.4节)。在*Ival_box*里用一个*BBwindow**表示窗口将导致另一种完全不同的设计，将引出另外的一组设计权衡问题 (12.7[14], 25.7节)。

有趣的是，*Ival_slider*的这个声明允许应用代码写得与原来一模一样。我们已经完成的所有东西就是将实现细节以一种更合乎逻辑的方式重新构造起来。

许多类都需要在它的对象消失之前做某些清理工作。因为抽象类*Ival_box*不知道其派生类

是否需要这类清理，它只能假定派生类需要做某些事情。我们保证正确清理的方式就是在基类里定义一个虚析构函数 `Ival_box::~~Ival_box()`，并在派生类里适当地覆盖它。例如，

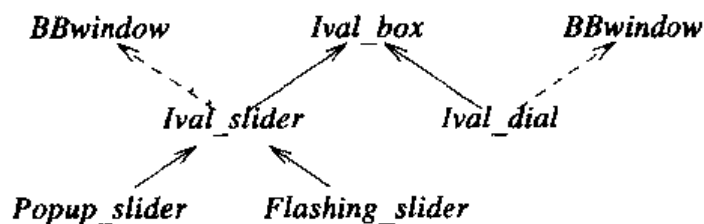
```
void f(Ival_box* p)
{
    // ...
    delete p;
}
```

将提示 `delete` 运算符去显式地销毁由 `p` 指向的对象。我们无法确知被 `p` 所指的对象到底属于哪个类，但是由于 `Ival_box` 的虚析构函数的作用，就会调用有关的类（可能有的）析构函数，执行其中所定义的正确清理工作。

`Ival_box` 的层次结构现在可以定义成下面的样子：

```
class Ival_box { /* ... */ };
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class Ival_dial : public Ival_box, protected BBwindow { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
```

可以有如下图示（经过明显的简化）



我用虚线表示 `protected` 继承。如果只考虑普通用户所关心的东西，它们所表示的不过是一些实现细节。

12.4.3 其他实现方式

与传统方式相比，这种设计更加清晰也更容易维护——而且并不低效。但无论如何，这种设计还是没有解决版本控制问题：

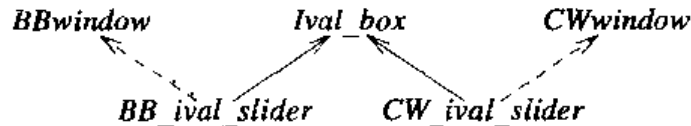
```
class Ival_box { /* ... */ }; // common
class Ival_slider : public Ival_box, protected BBwindow { /* ... */ }; // 为 BB
class Ival_slider : public Ival_box, protected CWwindow { /* ... */ }; // 为 CW
// ...
```

此外，也没有办法让用于 `BBwindow` 的 `Ival_slider` 与用于 `CWwindow` 的 `Ival_slider` 共存，即使那两个用户界面系统本身能够共存。

一个明显的解决方案是采用不同的名字，定义几个不同的 `Ival_slider` 类

```
class Ival_box { /* ... */ };
class BB_ival_slider : public Ival_box, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_box, protected CWwindow { /* ... */ };
// ...
```

图示是

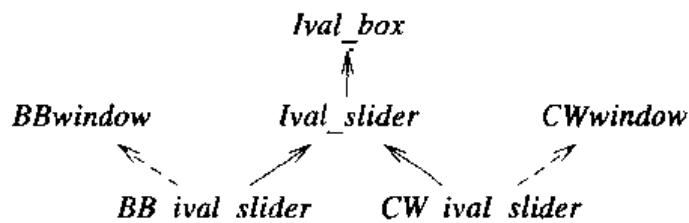


为进一步将我们面向应用的**Ival_box**类与实现细节隔离开，我们可以从**Ival_box**派生出一个抽象的**Ival_slider**类，而后从它派生出针对各个系统的**Ival_slider**类

```

class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class BB_ival_slider : public Ival_slider, protected BBwindow { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWwindow { /* ... */ };
// ...
  
```

图示是

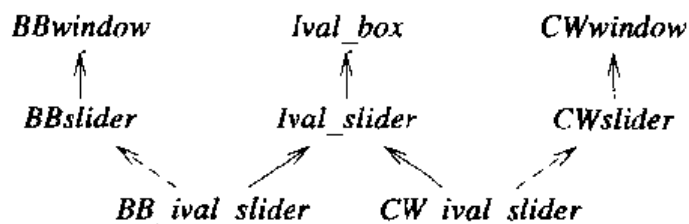


通常我们还可以利用实现层次方面的一些更特殊的类，把事情做得更好一些。例如，如果“Big Bucks Inc.”系统里有滑块类，我们就可以从**BBslider**派生我们的**Ival_slider**类

```

class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
  
```

图示是



在那些我们的抽象类与为实现而用的系统所提供的东西相差不远的地方（这也很常见），这一改进就非常重要。在这种情况下，在类似概念间的映射能大大减少程序设计工作。从比如**BBwindow**这样最一般的基类派生反而是非常罕见的。

完整的层次结构是将我们原来的面向应用的概念层次结构当做界面而组成的，用派生类表示：

```

class Ival_box { /* ... */ };
class Ival_slider : public Ival_box { /* ... */ };
class Ival_dial : public Ival_box { /* ... */ };
class Flashing_ival_slider : public Ival_slider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
  
```

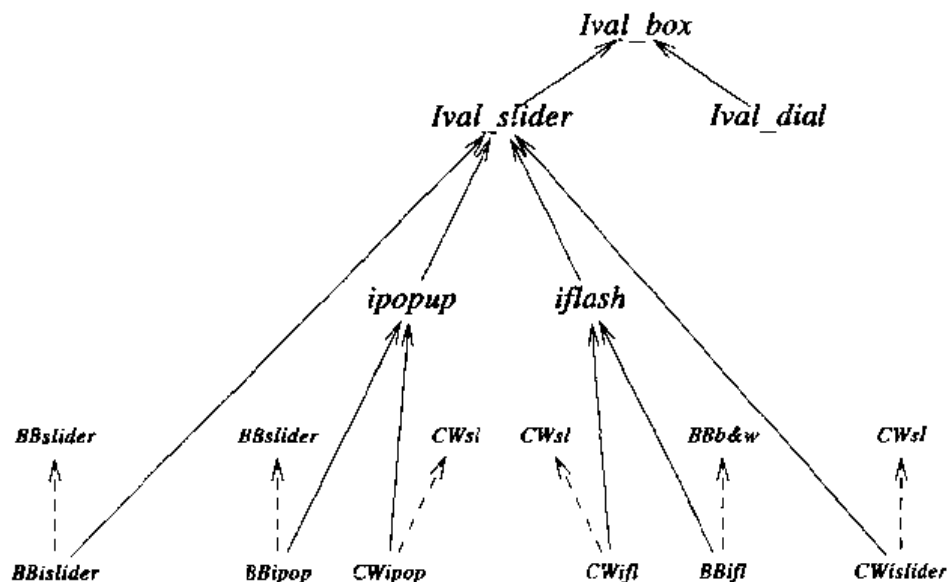
随后而来的是这一层次结构的针对各种图形用户界面系统的实现，也表述为派生类：

```

class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class BB_flashing_ival_slider : public Flashing_ival_slider,
  
```

```
protected BBwindow_with_bells_and_whistles { /* ... */ };
class BB_popup_ival_slider : public Popup_ival_slider, protected BBslider { /* ... */ };
class CW_ival_slider : public Ival_slider, protected CWslider { /* ... */ };
// ...
```

丢掉许多东西之后，这个层次结构可以图示为下面的样子：



原来的*Ival_box*类层次结构没有改变，但现在被许多实现类包围着。

12.4.3.1 评论

这种抽象类的设计非常灵活，而且，与那种依赖于一个定义用户界面系统的公共基类的等价设计方式比较，处理起来几乎同样容易。在后一种设计中，窗口类将成为一棵树的根。而对于前者，原来的应用类层次没有改变，成为所有类的根支持着整个应用的实现。从应用系统的角度看，这两种设计在某种很强的意义下是等价的：几乎所有代码都能不加修改，而且能以同样的方式使用。在这两种情况中，你在大部分时间都可以只看*Ival_box*一族类，而不必去担心与窗口有关的实现细节。例如，如果我们从一个类层次结构转移到另一个，我们应该不必去重写12.4.1节的*interact()*。

在两种情况中，如果用户界面系统的公用界面改变了，每个*Ival_box*类的实现都必须重写。但是，对于抽象类设计而言，几乎所有的用户代码都得到了保护，能够抵御实现层次上的结构变化，而且在这种变化之后不需要重新编译。当实现层次结构的提供商发布了一个“几乎兼容”的新版本时，这一点就显得特别重要了。除此之外，与传统层次结构的用户相比，抽象类层次结构的用户被锁定在某种专有实现上的危险性也更少一些。*Ival_box*抽象类应用层次结构的用户不会无意地使用任何来自实现的功能，因为只有在*Ival_box*层次结构中显式描述的功能才是可以访问的；这里没有从特定实现的基类隐式地继承来任何东西。

12.4.4 对象创建的局部化

一个应用的大部分可以利用*Ival_box*界面写出。进一步说，如果派生界面有所发展，提供了比普通*Ival_box*更多的功能，应用中的大部分就可以利用*Ival_box*、*Ival_slider*等界面写出来。但是在另一方面，对象的创建却必须通过特定于实现的名字去做，例如*CW_ival_dial*和

*BB_flashing_ival_slider*等。我们当然希望尽可能减少出现这些特殊名字的地方，但是对象的创建很难局部化，除非是采用某种系统化的方式去做。

与平常一样，解决问题的方式就是引进一个间接。这可以通过多种方式做到。一种简单方式就是引入一个抽象类，用以表示一组创建操作：

```
class Ival_maker {
public:
    virtual Ival_dial* dial(int, int) = 0;           // 做拨盘
    virtual Popup_ival_slider* popup_slider(int, int) = 0; // 做弹出式滑块
    // ...
};
```

对于来自*Ival_box*类族中的每一种用户应该知道的界面，类*Ival_maker*提供一个创建对应对象的函数。这样的类被称为是一个工厂，而它的函数有时（有点易于误解地）被称为虚构构造函数（15.6.2节）。

我们现在把每个用户界面系统表示为一个由*Ival_maker*派生出来的类：

```
class BB_maker : public Ival_maker {                // 做BB版本
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};

class LS_maker : public Ival_maker {                // 做LS版本
public:
    Ival_dial* dial(int, int);
    Popup_ival_slider* popup_slider(int, int);
    // ...
};
```

每个函数创建出具有所需界面和实现类型的一个对象。例如，

```
Ival_dial* BB_maker::dial(int a, int b)
{
    return new BB_ival_dial(a, b);
}

Ival_dial* LS_maker::dial(int a, int b)
{
    return new LS_ival_dial(a, b);
}
```

给定了一个到*Ival_maker*的指针，用户现在就可以创建对象了，然而却不必知道所用的究竟是哪个用户界面系统。例如，

```
void user(Ival_maker* pim)
{
    Ival_box* pb = pim->dial(0, 99);    // 创建适当的拨盘
    // ...
}

BB_maker BB_impl; // 为BB用户
LS_maker LS_impl; // 为LS用户

void driver()
{
```

```

    user(&BB_impl);    // 使用BB
    user(&LS_impl);    // 使用LS
}

```

12.5 类层次结构和抽象类

一个抽象类就是一个界面。类层次结构是一种逐步递增地建立类的方式。当然，每个类都为用户提供了一个界面，有些抽象类也提供了重要的功能，支持进一步向上构造。但是“界面”和“构造块”毕竟分别是抽象类和类层次结构的最本质角色。

类层次结构是一种层次结构，位于其中的各个类，一方面为用户提供了有用的功能，同时也作为实现更高级或者更特殊的类的构造砌块。这种层次结构对于支持以逐步求精方式进行的程序设计是非常理想的。它们为实现新的类提供了最大的支持，只要那些新类与现存的层次结构有着密切的关系。

传统的类层次结构倾向于在有关实现的考虑与提供给用户的界面之间建立起比较强的联系。抽象类可以在这里提供帮助。抽象类的层次结构是一种表述概念的清晰而强有力的方法，又不会用实现细节或者运行时的额外开销去妨碍这种表述。最后，虚函数调用是廉价的，与它所跨过的抽象边界的种类无关。调用一个抽象类的成员函数的开销与调用任何其他 *virtual* 函数完全一样。

按照这个方向考虑下去的逻辑结论就是：一个系统展现给用户的应该是一个抽象类的层次结构，其实现所用的是一个传统的层次结构。

12.6 忠告

- [1] 避免类型域；12.2.5节。
- [2] 用指针和引用避免切割问题；12.2.3节。
- [3] 用抽象类将设计的中心集中到提供清晰的界面方面；12.3节。
- [4] 用抽象类使界面最小化；12.4.2节。
- [5] 用抽象类从界面中排除实现细节；12.4.2节。
- [6] 用虚函数使新的实现能够添加进来，又不会影响用户代码；12.4.1节。
- [7] 用抽象类去尽可能减少用户代码的重新编译；12.4.2节。
- [8] 用抽象类使不同的实现能够共存；12.4.3节。
- [9] 一个有虚函数的类应该有一个虚析构函数；12.4.2节。
- [10] 抽象类通常不需要构造函数；12.4.2节。
- [11] 让不同概念的表示也不相同；12.4.1.1节。

12.7 练习

1. (*1) 定义

```

class Base {
public:
    virtual void iam() { cout << "Base\n"; }
};

```

从`Base`导出两个类，在这两个类中定义`iam()`，让它写出该类的名字。创建这些类的对象并对它们调用`iam()`函数。将指向派生类的对象的指针赋给`Base*`指针，而后通过这些指针调用`iam()`函数。

2. (*3.5) 利用你的系统中能用的所有图形功能实现一个简单的图形系统（如果你没有一个好图形系统，或者你对任何图形系统都没有经验，那么你也可以考虑一个简单的“大象素ASCII实现”，其中的一个点就是一个字符位置，你写时就是放一个适当的字符——例如 * ——在某个位置）：`Window(n, m)` 在屏幕上建立一块大小为`n`乘以`m`的区域。屏幕上的点用`(x, y)`坐标（笛卡儿坐标）作为地址。`Window w`有一个当前位置`w.current()`，初始时`current`是`Point(0, 0)`。当前位置可以用`w.current(p)`设置，其中的`p`是一个`Point`。一个`Point`用一对坐标刻画：`Point(x, y)`。一个`Line`用一对`Point`刻画：`Line(w.current(), p2)`。类`Shape`是`Dot`、`Line`、`Rectangle`、`Circle`等的公共界面。`Point`不是`Shape`。有一个`Dot`，`Dot(p)`用于表示在屏幕上的`Point p`。任何`Shape`都是不可见的，除非通过`draw()`将它画出来，例如：`w.draw(Circle(w.current(), 10))`。每个`Shape`有9个接触点：`e`（东）、`w`（西）、`n`（北）、`s`（南）、`ne`（东北）、`nw`（西北）、`se`（东南）、`sw`（西南）和`c`（中心）。例如，`Line(x.c(), y.nw())`将建立起一条从`x`的中心到`y`的左上角的线段。在用`draw()`画出一个`Shape`之后，当前位置就是这个`Shape`的`se()`。`Rectangle`用它的左下角和右上角刻画：`Rectangle(w.current(), Point(10, 10))`。作为一个简单试验，请设法显示出一个简单的“儿童画的房子”，带有一个房顶，两扇窗户和一个门。
3. (*2) 在屏幕上出现的`Shape`的一个重要方面是一组线段。实现一些操作去改变这些线段的表现形式：`s.thickness(n)`将线段的宽度设置为0、1、2或3，其中2为默认值，0表示不可见。此外，一个线段还可以是`solid`（实线）、`dashed`（短划线）和`dotted`（点线）。这通过函数`Shape::outline()`设置。
4. (*2.5) 提供函数`Line::arrowhead()`为线段末端加上箭头。一条线有两个端点，一个箭头可以指向相对于线段的两个方向。所以`arrowhead()`的参数必须能表示至少4个不同的值。
5. (*3.5) 保证落到`Window`外面的点或者线段不会出现在屏幕上。这通常称为“剪裁”。这里只作为一个练习，不要依靠实现所用的图形系统去做这件事。
6. (*2.5) 给图形系统添加`Text`类型。`Text`是矩形的显示字符的`Shape`。按默认方式，一个字符在每个坐标轴上都是一个坐标单位那么大。
7. (*2) 定义一个函数，它画出连接两个`Shape`的线段，方式是找出两个最接近的“接触点”，并连接起它们。
8. (*3) 为简单图形系统增加颜色概念。有三种东西可以着色：背景、封闭形状的内部、形状的轮廓线。
9. (*2) 考虑

```
class Char_vec {
    int sz;
    char element[];
public:
    static Char_vec* new_char_vec(int s);
    char& operator[](int i) { return element[i]; }
    // ...
};
```

定义 `new_char_vec()` 为 `Char_vec` 对象分配连续的存储区, 使得其中的元素可以像上面所示的那样用下标经过 `element` 访问。在什么环境中这个技巧会导致严重的问题?

10. (*2.5) 给定了从 `Shape` 类派生出的类 `Circle`、`Square` 和 `Triangle`, 定义函数 `intersect()`, 它以两个 `Shape*` 为参数, 调用适当的函数去确定这两个图形是否有重叠。为做这件事需要增加适当的(虚)函数。不必关心如何去写函数确定重叠问题, 只保证正确的函数能够被调用。这通常被称做双重指派或者多重方法。
11. (*5) 为写事件驱动的模拟设计和实现一个库。提示: `<task.h>`。当然, 这是一个很老的程序, 你可以做得更好些。这里应该有一个类 `task`。它的对象应该能保存自己的状态, 并能恢复这个状态(你可能需要定义 `task::save()` 和 `task::restore()`), 这使它可以像协作程序那样活动。特定的作业(task)可以定义为 `task` 类的派生类的对象。由一个 `task` 执行的程序可以用虚函数刻画。应该能将参数通过构造函数的参数传递给一个新作业。应该有一个调度器, 通过它实现虚拟时间的概念。提供函数 `task::delay(long)` 以“消费”虚拟时间。无论是把调度器作为作业类的一个部分, 还是将它分开做, 这都是一个重大决策。作业之间需要相互通信。请为通信设计一个 `queue` (队列) 类。设计出一种方式, 使作业能够等待来自若干队列的输入。用一种统一的方式处理运行时的错误。你将怎样查找和排除利用这个库写出的程序里的错误?
12. (*2) 为一个探险性游戏里的 `Warrior` (勇士)、`Monster` (怪兽) 和 `Object` (对象, 即那些你可以捡起、丢掉和使用等的东西) 类分别定义界面。
13. (*1.5) 为什么在 12.7[2] 中需要同时存在 `Point` 和 `Dot`? 在什么环境中将如 `Line` 等关键类的具体版本作为 `Shape` 类的参数是个好主意?
14. (*3) 按另一种实现策略勾画出 `Ival_box` 实例 (12.4节), 基于如下想法: 应用系统能看到的每个类是一个界面, 其中包含一个到实现的指针。这样, 每个“界面类”就成为实现类的一个句柄, 且同时存在着一个界面层次结构和一个实现层次结构。写出一些代码片段, 其中包含足够的细节, 以展示类型转换方面的可能问题。请考虑容易使用, 容易做程序设计, 在向层次结构中添加新概念时容易重用实现和界面, 容易进行界面和实现的修改, 以及在实现变化之后重新编译的必要性等问题。

第13章 模 板

你的引文在这里。

——B. Stroustrup

模板——字符串模板——实例化——模板参数——类型检查——函数模板——模板参数推断——描述模板参数——函数模板的重载——用模板参数描述策略——默认模板参数——专门化——派生和模板——成员模板——转换——源代码组织——忠告——练习

13.1 引言

独立的概念应该独立地表述，只在必要时才将它们组合起来。在违背这个原理的地方，你或者是将相互无关的概念捆到了一起，或者就是建立了不必要的依赖关系。无论哪种情况，你得到的都是一组更缺少灵活性的组件，并用它们去构造系统。模板提供了一种很简单的方式来表述范围广泛的一般性概念，以及一些组合它们的简单方法。这样产生出的类和函数，在运行时和空间效率方面，可以与手工写出的更特殊的代码媲美。

模板能直接支持通用型程序设计（2.7节），即那种采用类型作为参数的程序设计。C++的模板机制使人在定义类和函数时能以类型作为参数。模板只依赖于在实际使用时由它的参数而来的性质，并不要求被用做参数的那些不同类型之间有任何显式的联系。特别地，用于模板参数的这些类型不必来自同一个类层次结构。

这里对于模板的介绍将集中关注与标准库的设计、实现和使用有关的各种技术。与大部分软件相比，标准库要求的是更高水平的通用性、灵活性和效率。因此，能够用在标准库的设计和实现方面的各种技术，对于解决范围广泛的各种问题也必然是有效用的和高效率的。这些技术使实现者能够将复杂的实现隐藏在简单的界面背后，并使用户除了特别需要之外，都不会遭遇到这种复杂性。例如，`sort(v)`可能是许多不同排序算法的界面，这些算法能够用于存储着各种类型的元素的各种容器，对于某个特定的`v`将能自动选出最合适的排序算法。

每个主要的标准库抽象都被表示为一个模板（例如，`string`、`ostream`、`complex`、`list`和`map`等），所有的关键性操作（例如，`string`比较、输出运算符`<<`、`complex`的加法、取得`list`的下一个元素，以及`sort()`等）也是这样。这就使本书中有关库的各章（第三部分）成为模板实例和基于模板的程序设计技术的丰富资源。因此，本章将集中于一些较小的实例，用于阐释模板的各个技术侧面，以及使用它们的基本技术：

13.2节 定义和使用类模板的基本机制

13.3节 函数模板，函数重载和类型推断

13.4节 用模板参数刻画通用型算法的策略

13.5节 通过多个定义为一个模板提供多种实现

13.6节 派生和模板（运行时和编译时的多态性）

13.7节 源代码组织

模板已经在2.7.1节和3.8节介绍过。对于模板名解析、模板语法形式等的细节规则可以在C.13节中找到。

13.2 一个简单的String模板

考虑字符的串。串是一个能够保存一些字符的类，而且提供了我们通常认为与“串”这个概念有关的诸如下标选取、拼接和比较等操作。我们可能希望为多种不同种类的字符提供这样的功能。例如，有符号字符的串、无符号字符的串、中文字符的串、希腊文字符的串等，它们都是在某些环境中非常有用的东西。这样，我们就希望能够以最不依赖于特定字符种类的方式给出“串”这个概念。串的定义将依赖于字符可以复制这样一个事实，还有其他很少的东西。这样，我们就可以取来11.12节的`char`串，将其中的`char`类型做成参数，从而做出一个更具普遍性的串类型：

```
template<class C> class String {
    struct Srep;
    Srep *rep;
public:
    String();
    String(const C*);
    String(const String&);

    C read(int i) const;
    // ...
};
```

`template<class C>` 前缀说明当前正在声明的是一个模板，它有一个将在声明中使用的类型参数`C`。在引入之后，`C`的使用方式恰如其他的类型名。`C`的作用域将一直延伸到由这个`template<class C>` 作为前缀的声明的结束处。注意，`template<class C>` 只是说`C`是一个类型名，它不必一定是某个类的名字^①。

在类模板名字后随着由`<>`括起的一个类型名，也就成为（由这个模板所定义的）一个类的名字，可以像其他的类名字一样使用。例如，

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
    // 日文字符
};

String<Jchar> js;
```

除了名字的语法形式有些特殊之外，`String<char>` 的活动方式使它就像是由11.12节的`String`类定义出来的。将`String`做成一个模板，使我们能为任何种类的字符的`String`提供有关的功能，就像以前为`char`的`String`所做的那样。例如，如果我们使用标准库的`map`和这个`String`模板，11.8节的单词计数程序实例将变成：

① 类型名的概念比类名更宽泛，不仅包括由类定义引进的类类型的名字，还包括语言的内部类型、枚举、`typedef`引进的名字等。——译者注

```
int main()    // 统计输入中各个单词出现的次数
{
    String<char> buf;
    map<String<char>, int> m;
    while (cin>>buf) m[buf]++;
    // 写出结果
}
```

针对日文字符类型**Jchar**的版本就是：

```
int main()    // 统计输入中各个单词出现的次数
{
    String<Jchar> buf;
    map<String<Jchar>, int> m;
    while (cin>>buf) m[buf]++;
    // 写出结果
}
```

标准库提供了一个模板类**basic_string**，它很像这里模板化的**String**（11.12节、20.3节）。在标准库里，**string**被定义为**basic_string<char>**的同义词

```
typedef basic_string<char> string;
```

这就使我们可以写出如下的单词计数程序：

```
int main()    // 统计输入中每个单词出现的次数
{
    string buf;
    map<string, int> m;
    while (cin>>buf) m[buf]++;
    // 写出结果
}
```

一般来说，**typedef**在缩短由模板生成的长名字方面常常很有用处。另外，我们也经常希望不用去了解一个类型究竟是怎样定义的。**typedef**使我们可以隐藏起某个类型是由模板生成的这一事实。

13.2.1 定义一个模板

一个由类模板生成的类也是一个完全正常的类。这样，使用模板并不意味着需要任何超出等价的“手写”类所需要的运行时的机制，它也不必然地隐含着所产生的代码规模会有所减少。

首先做好特定的类，如**String**，并排除其中错误，而后再将它转为**String<C>**一类的模板，这种做法通常是个很好的主意。这样做，我们可以在具体实例的环境中处理好许多设计问题和大部分代码错误。所有程序员都熟悉这样的排除错误工作，大部分人应付具体实例的能力也胜过处理抽象概念。在此之后，我们就能将精力集中于处理由于推广而引起的问题，不会不断受到那些更平常的问题的滋扰。与此类似，当需要试图理解一个模板时，在试着去完全一般性地理解这个模板之前，针对一个特殊的类型参数（例如，**char**）想像它的行为方式也常大有裨益。

模板类中成员的声明和定义与在非模板类里完全一样。模板类的成员也不必在类本身的内部定义，在这种情况下，它的定义必须出现在某个地方，像非模板类的成员一样（C.13.7节）。

模板类的成员本身也是模板参数化的，与它们所在模板类的参数一样。在类外定义这些成员时，就必须显式地将它们定义为模板。例如，

```
template<class C> struct String<C>::Srep {
    C* s;           // 到元素的指针
    int sz;          // 元素个数
    int n;           // 引用计数
    // ...
};

template<class C> C String<C>::read(int i) const { return rep->s[i]; }

template<class C> String<C>::String()
{
    rep = new Srep(0, C());
}
```

像`C`这样的模板参数也是参数，而不是在模板之外有定义的类型名。然而，这不会影响用它们写模板代码的方式。在`String<C>`的作用域内部，用`<C>`作为给模板本身名字的限定词就多余了，所以`String<C>::String`就是构造函数的名字。如果你喜欢的话，当然也可以显式写出它来：

```
template<class C> String<C>::String<C>()
{
    rep = new Srep(0, C());
}
```

在一个程序里，对一个类成员函数只能有一个函数定义。同样，对于一个类模板成员函数，在一个程序里也只能有一个函数模板定义。但在另一方面，重载只能针对函数（13.3.2节）去做，而专门化（13.5节）使我们能够为同一个模板提供不同的实现。

类模板的名字不能重载。所以，如果在某个作用域里声明了一个类模板，就不能有其他以同样名字声明的实体（13.5节）。例如，

```
template<class T> class String { /* ... */ };
class String { /* ... */ }; // 错误：重复定义
```

作为一个模板的参数使用的类型，必须提供该模板所期望的界面。例如，作为`String`的参数使用的类型就必须提供常规的复制操作（10.4.4.1节、20.2.1节）。注意，在能够用于同一模板参数的不同类型之间并没有任何继承关系方面的要求。

13.2.2 模板实例化

从一个模板类和一个模板参数生成一个类声明的过程通常被称为模板实例化（C.13.7节）。这一说法也同样用于由模板函数加上模板参数产生（“实例化”）出一个函数的过程。针对一个特定模板参数的模板版本被称为是一个专门化。

一般来说，保证对所用的每组模板参数能够生成模板函数的相应版本的问题（C.13.7节）是具体实现的责任——而不是程序员的责任。例如，

```
String<char> cs;

void f()
{
```

```
String<Jchar> js;
cs = "It's the implementation's job to figure out what code needs to be generated";
}
```

对于这段代码，实现应该生成`String<char>`和`String<Jchar>`的声明、与它们对应的`Srep`类型、它们的析构函数和默认构造函数，以及赋值`String<char>::operator = (char*)`。其他成员函数没有被使用，因此就不应该生成。所生成出的类正是完全正常的类，遵守类的所有普遍有效的规则。与此类似，所生成的函数也是正常的函数，遵守函数的所有规则。

很明显，模板是一种从相对较短的定义生成出代码的强有力方法。因此，使用时也需要慎重，以免大量几乎相同的函数定义胀破了存储器（13.5节）。

13.2.3 模板参数

模板可以有类型参数，可以有常规类型的参数如`int`，还可以有模板参数（C.13.3节）。自然，一个模板可以有多个参数。例如，

```
template<class T, T def_val> class Cont { /* ... */ };
```

如上所示，一个模板参数可以用于定义跟随其后的模板参数。

整数参数常常被用于提供大小或者界限。例如，

```
template<class T, int i> class Buffer {
    T v[i];
    int sz;
public:
    Buffer() : sz(i) {}
    // ...
};

Buffer<char, 127> cbuf;
Buffer<Record, 8> rbuf;
```

当运行效率和紧凑性特别重要时（这些将阻止我们使用更具一般性的`vector`或`string`），像`Buffer`这样简单而受限的容器就可能很重要。将数组的大小作为参数传递，可以使`Buffer`的实现避免使用自由存储。另一个例子是25.6.1节的`Range`类型。

模板参数可以是常量表达式（C.5节），具有外部连接的对象或者函数的地址（9.2节），或者非重载的指向成员的指针（15.5节）。用做模板参数的指针必须具有`&of`的形式，其中`of`是对象或者函数的名字；或者具有`f`的形式，`f`必须是一个函数名。到成员的指针必须具有`&X::of`的形式，这里的`of`是一个成员名。特别地，字符串文字量不能被接受为模板参数。

整数模板参数必须是常量：

```
void f(int i)
{
    Buffer<int, i> bx;    // 错误：需要常量表达式
}
```

与此相对应，一个非类型的模板参数在模板的内部是一个常量，企图修改这种参数的值就是一个错误。

13.2.4 类型等价

给了一个模板，我们可以为它提供模板参数，以生成出一些类型。例如，

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>, 10> b1;
Buffer<char, 10> b2;
Buffer<char, 20-10> b3;
```

对于同一模板使用同一组模板参数，我们将总是表示同一个生成出的类型。但是，在这种情况下里的“同一组”是什么意思呢？通常，*typedef*并不引进新的类型，所以`String<Uchar>`是与`String<unsigned char>`同样的类型。与此相对应，因为`char`和`unsigned char`是不同的类型（4.3节），所以`String<char>`就是与`String<unsigned char>`不同的类型。

编译器能够对常量表达式求值（C.5节），所以`Buffer<char, 20-10>`和`Buffer<char, 10>`将被认为是同样的类型。

13.2.5 类型检查

模板首先被定义，而后与一组模板参数组合使用。在模板定义时，要检查这个定义的语法错误，也可能包括另外一些能独立于特定模板参数集而检查出来的错误。例如，

```
template<class T> class List {
    struct Link {
        Link* pre;
        Link* suc;
        T val;
        Link(Link* p, Link* s, const T& v) : pre(p), suc(s), val(v) { }
    } // 语法错误：缺少分号
    Link* head;
public:
    List() : head(7) { } // 错误：用int初始化指针
    List(const T& t) : head(new Link(0, 0, t)) { } // 错误：无定义标识符0
    // ...
    void print_all() const { for (Link* p = head; p; p=p->suc) cout << p->val << '\n'; }
};
```

编译器可以在定义点捕捉一些简单的语义错误，或者是以后在使用点再去做。用户通常希望能够早检查，但并不是所有“简单的”错误都很容易检查。在这里，我有意做进了三个“错误”。无论模板的参数是什么，指针`Link*`都不能用整数7去初始化。与此类似，标识符`0`（当然，是`0`输入错了）也不能作为`List<T>::Link`的构造函数的参数，因为作用域里没有这个名字。

在模板定义里所用的名字必须或者是在作用域里，或者以某种合理的明确方式依赖于某个模板参数（C.13.8.1节）。依赖于模板参数`T`的最常见最明显的方式是使用`T`的一个成员，或者有一个类型`T`的参数。在`List<T>::print_all()`里的`cout << p->val`是一个更微妙一些的例子。

与模板参数的使用有关的错误在模板使用之前不可能检查出来。例如，

```
class Rec { /* ... */ };

void f(const List<int>& li, const List<Rec>& lr)
```

```
{
    li.print_all();
    lr.print_all();
}
```

检查`li.print_all()`没有问题,但`lr.print_all()`将给出一个类型错,因为没有对`Rec`定义`<<`输出运算符。与模板参数相关的错误能被检查出来的最早位置,也就是在这个模板针对该特定模板参数的第一个使用点。这一点通常被称为第一个实例化点,简称实例化点(C.13.7节)。也允许具体实现将这种检查推迟到程序连接的时刻。如果我们在本编译单位里只能使用`print_all()`的声明,而不是它的定义,或许某个实现就必须推迟类型检查(见13.7节)。无论有关的检查在什么时候做,总是要检查同样的一组规则。再说一次,用户通常总希望及早检查。有可能通过成员函数来描述对模板参数的约束条件(13.9[6])。

13.3 函数模板

对于大部分人,首先的和最明显的模板应用就是定义和使用容器类,例如`basic_string`(20.3节),`vector`(16.3节),`list`(17.2.2节)和`map`(17.4.1节)等。不久就会出现对于模板函数的需求,数组排序是一个简单的例子:

```
template<class T> void sort(vector<T>&); // 声明

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi); // sort(vector<int>&);
    sort(vs); // sort(vector<string>&);
}
```

在调用模板函数时,函数参数的类型决定到底应使用模板的哪个版本,也就是说,模板的参数是由函数参数推断出来的(13.3.1节)。

自然,模板函数必须在某个地方定义(C.13.7节)

```
template<class T> void sort(vector<T>& v) // 定义
    // Shell sort (Knuth, Vol. 3, pg. 84).
{
    const size_t n = v.size();
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (v[j+gap]<v[j]) { // 交换v[j] 和v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

请将这个函数与7.7节的`sort()`做一个比较。这个参数化的版本更清楚,也更短,因为它能够依靠有关它所排序的元素类型的更多信息。它也很可能更快,因为它不是基于一个到比较函数的指针。这就意味着不需要间接的函数调用,而简单的`<`很容易在线化。

进一步的简化是利用标准库模板`swap()`(18.6.8节)将动作约减到更自然的形式

```
if (v[j+gap]<v[j]) swap(v[j], v[j+gap]);
```

这样做不会带来任何新的开销。

在这个例子里是用运算符 `<` 做比较。然而，并不是每个类型都有 `<` 运算符。这也就限制了 `sort()` 的这个版本的使用。当然，这种限制是很容易避免的（13.4节）。

13.3.1 函数模板的参数

模板函数对于写出通用型算法是至关重要的，这种算法可以应用于广泛且多样化的容器类型（2.7.2节、3.8节，第18章）。对于一个调用，能从函数的参数推断出模板参数的能力是其中最关键的东西。

编译器能够从一个调用推断出类型参数和非类型参数，条件是由这个调用的函数参数表能够惟一地标识出模板参数的一个集合（C.13.4节）。例如，

```
template<class T, int i> T& lookup(Buffer<T,i>& b, const char* p);

class Record {
    const char v[12]
    // ...
};

Record& f(Buffer<Record,128>& buf, const char* p)
{
    return lookup(buf,p); // 使用lookup(), 其中T是Record, i是128
}
```

在这里，推断出 `T` 是 `Record`，而且 `i` 是 `128`。

注意，绝不会对类模板的参数做任何推断。究其原因，对一个类可以提供多个构造函数，这种灵活性将会使有关的推断在许多情况下都无法完成，在更多情况下是不清楚的。专门化是为在一个类的不同实现之间做隐式选择而提供的一种机制（13.5节）。如果需要创建推断出的某类型的一个对象，我们常需要通过调用一个能完成这个创建的函数来做这件事；参看17.4.1.2节的 `make_pair()`。

如果不能从模板函数的参数推断出某个模板参数（C.13.4节），我们就必须显式地去描述它。做这件事的方式与显式地为模板类提供类型参数一样。例如，

```
template<class T> class vector { /* ... */ };
template<class T> T* create(); // 做一个T, 返回到它的指针

void f()
{
    vector<int> v;           // 类、模板参数'int'
    int* p = create<int>();  // 函数、模板参数'int'
}
```

显式描述的最常见用途是为模板函数提供返回值类型：

```
template<class T, class U> T implicit_cast(U u) { return u; }

void g(int i)
{
    implicit_cast(i);           // 错误：无法推断T
    implicit_cast<double>(i);   // T是double, U是int
    implicit_cast<char,double>(i); // T是char, U是double
    implicit_cast<char*,int>(i); // T是char*, U是int; 错误：不能将int转换为char*
}
```

就像默认的函数参数一样（7.5节），在显式模板参数表中，只有位于最后的类型可以不给出。

由于可以显式地描述模板参数，这就使我们能够定义各种转换函数族和对象创建函数（13.3.2节、C.13.1节和C.13.5节）。*implicit_cast*()是隐式转换的一个显式版本（C.6节），这种东西经常很有用处。*dynamic_cast*、*static_cast*等（6.2.7节、15.4.1节）的语法形式与显式限定的模板函数的语法形式相匹配。当然，这些内部类型转换运算符所提供的操作是无法通过其他语言特征描述的。

13.3.2 函数模板的重载

可以声明多个具有同样名字的函数模板，甚至可以声明具有同一个名字多个函数模板和常规函数的组合。当一个重载了的函数被调用时，就需要通过重载解析去找出应该调用的正确函数或者模板函数。例如，

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
    sqrt(2);           // sqrt<int>(int)
    sqrt(2.0);         // sqrt(double)
    sqrt(z);           // sqrt<double>(complex<double>)
}
```

就像模板函数是函数概念的推广一样，出现了函数模板时的解析规则也是对重载函数解析规则的推广。简而言之，对每个模板，我们都要找出对这组函数参数的一组最佳专门化结果，而后我们将普通函数的重载解析规则应用到这些专门化结果和所有常规函数上：

- [1] 找出能参与这个重载解析的一组函数模板的专门化（13.2.2节）。做这件事情的方式就是查看每一个函数模板，在假定没有其他同名函数模板或函数在作用域里的条件下，确定对它是否存在一组可以使用的模板参数，如果存在的话究竟是哪一组参数。例如对于调用*sqrt(z)*，这将产生出候选函数*sqrt<double>(complex<double>)*和*sqrt<complex<double> >(complex<double>)*。
- [2] 如果两个模板函数都可以调用，而其中的一个比另一个更专门（13.5.1节），在随后的步骤中就只考虑那个最专门的模板函数。对于调用*sqrt(z)*，这意味着应该选*sqrt<double>(complex<double>)*而非*sqrt<complex<double> >(complex<double>)*，因为任何与*sqrt<T>(complex<T>)*匹配的调用也一定能与*sqrt<T>(T)*匹配。
- [3] 在这组函数上做重载解析，包括那些按照常规函数考虑（7.4节）也应该加上去的常规函数。如果某个模板函数参数已经通过模板参数推断确定下来（13.3.1节），这个参数就不能再同时应用提升、标准转换或者用户定义的转换。对于*sqrt(2)*，*sqrt<int>(int)*是确切匹配，它当然优先于*sqrt(double)*。
- [4] 如果一个函数和一个专门化具有同样好的匹配，那么就选用函数。因为这个原因，对于*sqrt(2.0)*将选用*sqrt(double)*而不是*sqrt<double>(double)*。
- [5] 如果找不到匹配，这就是一个错误。如果最后得到了两个或者更多同样好的匹配，这个调用就是有歧义的，是一个错误。

例如，


```

template<class T> T max(T, T);

const int s = 7;

void k()
{
    max(1, 2);           // max<int> (1,2)
    max('a', 'b');       // max<char> ('a','b')
    max(2.7, 4.9);       // max<double> (2.7,4.9)
    max(s, 7);           // max<int>(int(s), 7) (用简单转换)

    max('a', 1);         // 错误: 有歧义 (没有标准转换)
    max(2.7, 4);         // 错误: 有歧义 (没有标准转换)
}

```

我们可以通过显式限定来消解这两个歧义

```

void f()
{
    max<int>('a', 1);     // max<int> (int('a'),1)
    max<double>(2.7, 4);  // max<double> (2.7,double(4))
}

```

或者是增加适当的声明

```

inline int max(int i, int j) { return max<int>(i, j); }
inline double max(int i, double d) { return max<double>(i, d); }
inline double max(double d, int i) { return max<double>(d, i); }
inline double max(double d1, double d2) { return max<double>(d1, d2); }

void g()
{
    max('a', 1); // max(int('a'),1)
    max(2.7, 4); // max(2.7,double(4))
}

```

对常规函数就使用常规的重载规则(7.4节), *inline*的使用可以保证不会强加任何额外开销。

max() 的定义很简单, 所以我们原本就可以显式地写出它来。无论如何, 采用模板专门化是定义这种解析功能的简单而通用的方式。

重载解析规则保证模板函数能够正确地与继承机制相互作用

```

template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T> *);

void g(B<int> * pb, D<int> * pd)
{
    f(pb); // f<int> (pb)
    f(pd); // f<int> (static_cast<B<int>*>(pd)); 采用D<int>* 到B<int>* 的标准转换
}

```

在这个例子里, 模板函数 *f()* 对任何类型 *T* 都能接受 *B<T>**。我们有一个类型为 *D<int>** 的参数, 因此编译器很容易通过将 *T* 选为 *int*, 从而做出推断, 这个调用能被惟一地解析为一个对 *f(B<int>*)* 的调用。

在对模板参数的推断中未涉及到的那些函数参数, 就完全按照非模板函数的参数处理。特别是可以使用普通的转换规则。考虑

```
template<class T, class C> T get_nth(C& p, int n); // 取出第n个元素
```

假定这个函数返回类型为C的容器里的第n个元素的值。因为在一个调用中，类型C必须在调用中从get_nth()的实际参数中推断出来，所以对于其中的第一个参数就不能做转换。但是第二个参数完全是正常的，所有可能的转换都需要考虑。看下面例子：

```
class Index {
public:
    operator int();
    // ...
};

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth<int>(v, 2);    // 精确匹配
    int i2 = get_nth<int>(v, s);    // 标准转换: short到int
    int i3 = get_nth<int>(v, i);    // 用户定义转换: Index到int
}
```

13.4 用模板参数描述策略

现在考虑如何做串的排序。这里牵涉到三个概念：串、元素类型，还有排序算法用于比较串元素的准则。

我们无法将排序的准则硬编码在容器里，因为容器（一般说）不能对它的元素类型强加上自己的需求。我们也无法将排序准则硬编码到元素类型里，因为有可能存在多种不同的对元素排序的方式。

因此，排序准则就既不能构筑在容器里，也不能构筑进元素类型里。与此对应，只能在需要做特定操作的时候提供所用的排序准则。举个例子，假定我用字符串表述瑞典人的名字，那么应该用什么样的字符比较准则呢？瑞典人名的排序中常使用的有两种字符序列（字符的编号顺序）。很自然，任何通用的串类型或者通用的排序程序都不可能知道瑞典人名的排序习惯。所以，任何一种通用的解决方案，都要求排序算法应当以某种通用的方式表述，使它的定义不但可以用于特定类型，而且还可以用于特定类型的特定使用。例如，让我们把标准C库函数strcmp()推广到任意类型T的String（13.2节）：

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i<str2.length(); i++)
        if (!C::eq(str1[i], str2[i])) return C::lt(str1[i], str2[i]) ? -1 : 1;
    return str1.length() - str2.length();
}
```

如果什么人想让compare()忽略大小写、反应本地习惯，如此等等，通过适当地定义C::eq()和C::lt()都可以办到。这就使任何（比较、排序等）算法都可以在由“C操作”提供的操作和所描述的容器的基础上表述清楚。例如，

```
template<class T> class Cmp { // 常规，默认比较
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
```

```
};

class Literate { // 根据文化习惯比较瑞典人的名字
public:
    static int eq(char a, char b) { return a==b; }
    static int lt(char, char); // 基于字符值查一个表 (13.9[14])
};
```

现在我们就可以通过模板参数的显式地描述用于比较的规则了:

```
void f(String<char> swede1, String<char> swede2)
{
    compare< char, Cmp<char> > (swede1, swede2);
    compare< char, Literate > (swede1, swede2);
}
```

与其他方式相比, 例如把指针传递给函数, 将比较操作作为模板参数传递有两个重要的优点: 可以通过一个参数传递几个操作, 又没有任何运行时开销。比较操作`eq()`和`lt()`很容易在线化, 而要把通过到函数的指针所做的调用在线化, 编译器就需要特别当心了。

很自然, 不但可以为用户定义类型提供比较操作, 对内部类型也可以这样做。为使通用算法能将非平凡的比较准则应用到各种类型上, 上述性质是最根本的基础 (18.4节)。

从一个类模板生成的每个类都将得到类模板中每个`static`成员的一个副本 (C.13.1节)。

13.4.1 默认模板参数

每次调用都需要显式地给出比较准则也会使人厌烦。还好, 很容易确定一种默认方式, 这样, 就只有那些不平常的比较准则才需要显式地给出。这可以通过重载实现:

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2); // 用C比较

template<class T>
int compare(const String<T>& str1, const String<T>& str2); // 用Cmp<T> 比较
```

换一种方式, 我们也可以将常规的习惯作为默认模板参数:

```
template<class T, class C = Cmp<T> >
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i<str2.length(); i++)
        if (!C::eq(str1[i], str2[i])) return C::lt(str1[i], str2[i]) ? -1 : 1;
    return str1.length() - str2.length();
}
```

有了这些之后, 我们就可以写:

```
void f(String<char> swede1, String<char> swede2)
{
    compare(swede1, swede2); // 用Cmp<char>
    compare<char, Literate>(swede1, swede2); // 用Literate
}
```

一个 (对于非瑞典人而言) 不那么深奥的例子是在比较时考虑或忽略大小写:

```
class No_case { /* ... */ };

void f(String<char> s1, String<char> s2)
```

```

{
    compare(s1, s2);           // 区分大小写
    compare<char, No_case>(s1, s2); // 忽略大小写
}

```

通过模板参数支持执行策略，以及在此基础上采用默认值支持最常用策略的技术在标准库中广泛使用（例如，18.4节）。很奇怪，这种技术没有用于做`basic_string`比较（13.2节、第20章）。用于表述策略的模板参数常常被称为“特征（traits）”。例如，标准库`string`依赖于`char_traits`（20.2.1节），标准算法依赖于迭代器特征类（19.2.2节），所有标准库容器依赖于`allocator`（19.4节）。

对一个模板参数的默认实际参数的语义检查是在这个默认值实际被使用之时（且只在此时）。特别地，只要一直避免使用默认的模板实际参数`Cmp<T>`，我们就可以用`compare()`比较类型`X`的串，即使`Cmp<X>`不能编译（比如说，`X`里没有`<`的定义）。这一点对于设计标准容器极其重要，这些容器都依赖于给定模板参数默认值的技术（16.3.4节）。

13.5 专门化

按照默认约定，一个模板给出了一个单一的定义，它可以用于用户可以想到的任何模板参数（或者模板参数组合）。对于某些写模板的人而言，这个规定不一定总有意义。我可能想说“如果模板参数是指针，就用这个实现；如果不是，就用那个实现”。或者说“除非模板参数是从类`My_base`派生的指针，否则就是一个错误”。许多这样的设计考虑都可以通过提供多个不同定义的方式来处理，并由编译器基于在使用处所提供的模板参数，在这些定义中做出选择。对一个模板的这些可以互相替代的定义称为用户定义的专门化，或简称为用户专门化。

考虑一个`Vector`模板的可能使用情况

```

template<class T> class Vector {    // 通用向量类
    T* v;
    int sz;
public:
    Vector();
    explicit Vector(int);

    T& elem(int i) { return v[i]; }
    T& operator[] (int i);

    void swap(Vector&);
    // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;

```

大部分`Vector`将是某种指针类型的`Vector`。这样做的原因很多，但最基本的原因是，为了保持多态性的行为方式就必须使用指针（2.5.4节、12.2.6节）。也就是说，任何人要实践面向对象的程序设计，又要使用类型安全的容器（例如，标准库里的容器），最后总会导致许许多多的指针容器。

大部分C++ 实现的默认方式是对模板函数建立代码段的副本。对于运行效率而言, 这种做法很好。但如果我们不小心从事, 在许多关键情况下它也会导致严重的代码膨胀, 就像在上面`Vector`的例子中。

还好, 存在着一种明显的解决办法, 可以使所有的指针容器都能共享同一份实现代码。这件事可以通过专门化做到。首先, 我们定义一个到`void`的指针的(专门化的)`Vector`版本

```
template<> class Vector<void*> {
    void** p;
    // ...
    void*& operator [] (int i);
};
```

而后这个专门化就会被用做所有指针的`Vector`的共同实现了。

前缀`template<>` 说明这是一个专门化, 可以不用模板参数描述。这个专门化使用时的模板参数在名字之后的 `<>` 括号里描述。也就是说, `<void*>` 说明这个定义应该用在所有的`T`是`void*` 的`Vector`实现里。

`Vector<void*>` 是一个完全的专门化, 也就是说, 在我们使用这个专门化时根本不需要描述或者推断模板参数。`Vector<void*>` 将用于如下定义的`Vector`:

```
Vector<void*> vpv;
```

要定义一个专门化, 使它能够用于所有的指针的`Vector`, 且仅仅用于指针的`Vector`, 我们就需要一个部分专门化:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;
    Vector() : Base() {}
    explicit Vector(int i) : Base(i) {}
    T*& elem (int i) {return reinterpret_cast<T*&>(Base::elem (i)); }
    T*& operator [] (int i) {return reinterpret_cast<T*&>(Base::operator [] (i)); }
    // ...
};
```

在名字之后的 `<T*>` 说明这个专门化将被用于每一个指针类型。换句话说, 这个定义将用于每一个其模板参数能够表述为`T*` 的`Vector`。例如,

```
Vector<Shape*> vps; // <T*> 是 <Shape*>, 所以T是Shape
Vector<int**> vppi; // <T*> 是 <int**>, 所以T是int*
```

注意, 当用到某个部分专门化时, 模板参数是从专门化模式推断出来的, 该模板参数并不是一定就是那个模板实际参数。特别地, 对于`Vector<Shape*>`, `T`是`Shape`而不是`Shape*`。

有了`Vector`的这个部分专门化之后, 我们就使所有指针的`Vector`共享了同一个实现。`Vector<T*>` 实际上只是一个到`Vector<void*>` 的界面, 它完全是通过派生和在线展开实现的。

有一点非常重要: 取得对`Vector`实现的这种精化, 并没有影响呈现给用户的界面。专门化是一种针对一个共同界面, 为不同的使用提供各种可替代的实现的方法。当然, 我们也可以给通用`Vector`和指针的`Vector`取不同的名字。然而, 如果我真的那样做了, 许多对详情不大了解的人就会忘记这个专为指针而做的类, 并发现他们的代码比自己所期望的大了很多。由于

这种情况，更好的做法就是将关键性的实现细节隐藏在一个公共界面之后。

实际使用已经证明，这种技术在抑制代码膨胀方面非常有效。没有使用类似技术的人们（在C++或者其他提供了类型参数化机制的语言里）常常发现，即使对中等规模的程序，代码副本也可能消耗掉成兆字节的代码空间。用一个专门化去实现所有指针的表可以作为例子，说明一种通过最大限度的代码共享达到最少的代码膨胀的一般性技术。

通用模板必须在所有专门化之前声明。例如，

```
template<class T> class List<T*> { /* ... */ };
template<class T> class List { /* ... */ }; // 错误：通用模板在专门化之后
```

通用模板所提供的最关键信息就是那一组模板参数。用户在使用这个模板时，在做它的专门化时都必须提供这些参数：

```
template<class T> class List;
template<class T> class List<T*> { /* ... */ };
```

如果需要使用的话，通用模板就必须在某个地方有定义^①（13.7节）。

如果用户在某处对一个模板进行了专门化，那么，在针对与此专门化相关的类型使用这个模板的每个位置，这个专门化都必须在作用域里。例如，

```
template<class T> class List { /* ... */ };
List<int*> li;
template<class T> class List<T*> { /* ... */ }; // 错误
```

这里是在`List<int*>`用过之后，又做了`List`对`int*`的专门化。

一个模板的所有专门化和这个模板本身都必须在同一个名字空间里声明。如果用到某个显式声明了的专门化，那么它就必须要在某个地方显式定义（与通过更一般的模板生成不同，13.7节）。换句话说，对模板做显式的专门化，隐含着不再为这个专门化生成定义。

13.5.1 专门化的顺序

说一个专门化比另一个更专门，如果能够与它匹配的每个实际参数表也能与另外的那个专门化匹配，但反过来就不对。例如，

```
template<class T> class Vector;           // 一般情况
template<class T> class Vector<T*>;      // 对任何指针的专门化
template<> class Vector<void*>;          // 对void*的专门化
```

每个类型都可以作为最一般的`Vector`的实际参数，只有指针能够作为`Vector<T*>`的实际参数，而只有`void*`才能作为`Vector<void*>`的实际参数。

在声明对象、指针等的时候（13.5节），在做重载解析的时候（13.3.2节），最专门化的那个版本都将比别的版本优先被选中。

一个专门化模式可以通过一些类型来描述，这些类型可以是由模板参数推断中允许使用的那些结构（13.3.1节）组合而成的。

① 注意，作者常常采用这种说法，其意思就是：在使用的地方，必须能够“看到”这个通用模板的定义，仅仅能看到它的声明是不够的。其他地方也是同样意思。——译者注

13.5.2 模板函数的专门化

很自然,专门化对于模板函数也非常有意义。考虑7.7节和13.3节的Shell排序,它用 < 比较元素,并用细节代码完成元素交换。一个更好的定义可能是

```
template<class T> bool less(T a, T b) { return a<b; }

template<class T> void sort(Vector<T>& v)
{
    const size_t n = v.size();
    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap], v[j])) swap(v[j], v[j+gap]);
}
```

这样做并没有改进算法本身,但它却使对实现的改进成为可能。但是按这种写法, `sort()` 将无法正确完成对 `Vector<char*>` 的排序,因为 < 将会去比较两个 `char*`。也就是说,它去比较两个串中第一个字符的地址。与此相反,我们当然希望它能去比较被指向的那些字符。用 `less()` 对 `const char*` 的一个简单的专门化就能做好这件事:

```
template<> bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a, b) < 0;
}
```

与类一样(13.5节), `template<>` 前缀说明这是一个专门化,在描述时不用模板参数。在模板函数名之后的 `<const char*>` 说明这个专门化应该在模板参数是 `const char*` 的情况下使用。由于模板参数可以从函数的实际参数表推断,所以我们不需要显式地去描述它。因此,我们就可以简化这个专门化的定义:

```
template<> bool less<>(const char* a, const char* b)
{
    return strcmp(a, b) < 0;
}
```

给出了 `template<>` 前缀,第二个空的 <> 也就多余了,因此我们常常简单地写:

```
template<> bool less(const char* a, const char* b)
{
    return strcmp(a, b) < 0;
}
```

我喜欢这种简短的声明形式。

考虑 `swap()` 的明显定义方式:

```
template<class T> void swap(T& x, T& y)
{
    T t = x;        // 把x复制到临时量
    x = y;           // 把y复制到x
    y = t;           // 把临时量复制到y
}
```

在对 `Vector` 的 `Vector` 调用时,这种方式相当低效,因为它将通过复制全部元素的方式交换 `Vector`。这个问题也可以通过适当的专门化解决。`Vector` 对象本身只保存着足以提供对元素间

接访问的信息（就像String那样，11.12节、13.2节）。因此，对Vector的交换就可以通过交换其表示来完成。为了能够对这种表示进行操作，我为Vector类提供了一个成员函数swap()（13.5节）：

```
template<class T> void Vector<T>::swap(Vector & a)    // 交换表示
{
    swap(v, a.v);
    swap(sz, a.sz);
}
```

现在就可以利用这个成员来定义一般性swap()的专门化了

```
template<class T> void swap(Vector<T>& a, Vector<T>& b)
{
    a.swap(b);
}
```

对less()和swap()的这些专门化都用在标准库里（16.3.9节、20.3.16节）。另外，它们也是具有广泛适用性的技术的实例。如果对一组实际模板参数的一般性算法（这里是swap()），存在着某些效率更高的特殊的替代品，专门化就会很有价值。除此之外，当某种实际参数类型由于其不规范性，导致一般性算法给出我们所不希望的结果时（这里的less()），通过专门化解决也很方便。这种“不规则类型”常常是内部的指针或者数组类型。

13.6 派生和模板

模板和派生都是从已有类型构造新类型的机制，通常都被用于去写利用各种共性的代码。如3.7.1节、3.8.5节和13.5节所示，这两种机制的组合是许多有用技术的基础。

从一个非模板类派生出一个模板类，这是为一组模板提供一个共用实现的一种方法。13.5节的向量是这一方法的很好示例：

```
template<class T> class Vector<T*> : private Vector<void*> { /* ... */ };
```

对这种示例还可以有另一种看法，这里用一个模板类为某种功能提供了一个优美而且类型安全的界面，而该功能原本是不安全的，使用起来也很不方便。

很自然，从一个模板类派生出另一个模板类常常也很有用。基类的一种用途就是作为实现其他类的构造块。如果某个基类的成员依赖于其派生类的模板参数，那么这个基类本身也必须参数化。3.7.2节的Vec是一个这方面的示例

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

对模板函数的重载解析规则保证，这些函数一定能为派生类型“正确地”工作（13.3.2节）。

最常见的情况是基类和派生类具有同样的模板参数，但这并不是一项要求。确实有一些虽然不太常见但也很有趣的技术，依赖于将派生类型本身传递给基类。例如，

```
template <class C> class Basic_ops { // 容器的基本运算符
public:
    bool operator==(const C&) const;    // 比较所有元素
    bool operator!=(const C&) const;
    // ...
    // 给C操作访问权
    const C& derived() const { return static_cast<const C&>(*this); }
```



```
};

template<class T> class Math_container : public Basic_ops< Math_container<T> > {
public:
    size_t size() const;
    T& operator[] (size_t);
    const T& operator[] (size_t) const;
    // ...
};
```

这就使有关容器的基本操作定义可以与容器本身的定义分开，而且只需要定义一次。然而，像 `==` 或 `!=` 这种运算的定义必须基于容器和其元素两者来描述，所以需要将该元素类型传递给容器模板，然后将作为结果的容器类型传递给 `Basic_ops`。

假定这个 `Math_container` 类似于传统的向量，`Basic_ops` 成员的定义就应该像下面这种样子：

```
template <class C> bool Basic_ops<C>::operator==(const C& a) const
{
    if (derived().size() != a.size()) return false;
    for (int i = 0; i < derived().size(); ++i)
        if (derived()[i] != a[i]) return false;
    return true;
}
```

保持容器与操作分离的另一种替代技术是通过模板参数将它们组合起来，而不用经过派生：

```
template<class T, class C> class Mcontainer {
    C elements;
public:
    T& operator[] (size_t i) { return elements[i]; }

    friend bool operator==(<> (const Mcontainer&, const Mcontainer&); // 比较元素
    friend bool operator!=(<> (const Mcontainer&, const Mcontainer&);
    // ...
};

template<class T> class My_array { /* ... */ };

Mcontainer< double, My_array<double> > mc;
```

通过类模板生成的类是完全正常的类，因此，它们也可以有 `friend` 函数（C.13.2节）。在这一例子里我使用了 `friend` 函数，以便得到符合习惯的 `==` 和 `!=` 的对称性参数风格（11.3.2节）。在这些情况中，人们也可以考虑将一个模板作为 `C` 的实际参数传递，而不是去传递一个容器（C.13.3节）。

13.6.1 参数化和继承

一个模板将一个类型或者一个函数用其他的类型进行参数化。该模板对于所有参数类型的代码实现都是相同的，因为大部分代码来自于这个模板。一个抽象类定义了一个界面，使这个抽象类的不同实现中的许多代码可以由该类层次结构所共享，而使用这个抽象类的大部分代码都不依赖于它的实现。从某种设计观点看，这两种方式如此接近，因此应该有一种共同的称谓。因为两者都使我们能只表述一次，而后用于许多不同的类型，人们有时用多态性来描述它们。为了能有所区分，将虚函数提供的东西称做运行时多态性，而把模板提供的称

为编译时多态性或者参数式多态性。

那么，什么时候我们应该选用模板，什么时候应该依靠一个抽象类呢？在这两种情况下，我们都是操纵一些共享着一组操作的对象。如果在这些对象间并不需要某种层次性的关系，那么最好是将它们作为模板的参数。如果在编译时无法确知这些对象的类型，那最好是将它们表示为一个公共抽象类的许多派生类。如果对运行时效率的要求特别严格，也就是说，必须将各种操作在线化，那么就应该使用模板。24.4.1节将讨论这个问题的更多细节。

13.6.2 成员模板

一个类或者模板也可以包含本身就是模板的成员。例如，

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    template<class T>
        complex(const complex<T>& c) : re(c.real()), im(c.imag()) {}
    // ...
};

complex<float> cf(0,0);
complex<double> cd = cf; // 可以：用float到double的转换

class Quad {
    // 没有到int的转换
};

complex<Quad> cq;
complex<int> ci = cq; // 错误：从Quad到int的转换
```

换句话说，当且仅当你能够用T2对T1进行初始化，你就可以从一个complex<T2>构造出一个complex<T1>。这看起来很合理。

不幸的是，C++ 继承了一些在内部类型之间的不甚合理的转换，例如从double到int。采用implicit_cast（13.3.1节）或者checked（C.6.2.6节）方式的检查转换，就能够捕获这种在运行中发生的截断问题：

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    complex() : re(0), im(0) {}
    complex(const complex<Scalar>& c) : re(c.real()), im(c.imag()) {}

    template<class T2> complex(const complex<T2>& c)
        : re(checked_cast<Scalar>(c.real())), im(checked_cast<Scalar>(c.imag())) {}
    // ...
};
```

为了完全起见，我在这里增加了一个默认构造函数和一个复制构造函数。非常奇怪，模板构造函数绝不会被用于生成复制构造函数，因此，如果不显式地声明一个复制构造函数，那就会自动生成一个默认的复制构造函数。在这一情况下，生成出来的那个复制构造函数应该与我明确给出的完全一样。类似地，复制赋值（10.4.4.1节、11.7节）也必须定义为非模板运算符。

成员模板不能是virtual。例如，

```
class Shape {
    // ..
```

```
template<class T> virtual bool intersect(const T&) const =0; // 错误: virtual模板
};
```

这必须是非法的。如果我们允许这样,用于实现虚函数的常规虚函数表技术(2.5.5节)就无法使用了。因为,当`intersect()`每次用一个新的实参类型调用时,连接程序就必须向`Shape`类的虚表里加入一个表项。

13.6.3 继承关系

将类模板理解为一种有关如何生成特定类的规范,这也是一种很有用的看法。换句话说,模板实现的是一种在需要时能够基于用户描述去生成类型的机制。正因为此,类模板有时也被称为类型生成器。

如果只考虑C++语言的规则,在由同一个类模板生成的两个类之间并不存在任何关系。例如,

```
class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

有了这些声明之后,人们有时会想到把`set<Circle*>`当做`set<Shape*>`。这是来自如下有缺陷论证的一个严重逻辑错误:“一个`Circle`是一个`Shape`,所以一个`Circle`的集合也就是一个`Shape`的集合。因此,我应该能把`Circle`的集合当做`Shape`的集合”。这个论证的“所以”部分并不成立。究其原因,一个`Circle`的集合将保证它的成员都是`Circle`,而`Shape`的集合就不能提供这种保证^①。例如,

```
class Triangle : public Shape { /* ... */ };

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle());
    // ...
}

void g(set<Circle*>& s)
{
    f(s); // 错误; 类型不匹配: s是set<Circle*>, 不是set<Shape*>
}
```

这将无法编译,因为不存在内部的从`set<Circle*>&`到`set<Shape*>&`的转换,也不应该有这种转换。保证`set<Circle*>`的成员一定是`Circle`,这将使我们能安全高效地对这个集合的成员应用`Circle`的特殊操作,如确定其半径值等。如果允许将`set<Circle*>`当做`set<Shape*>`看待,我们就不再有这种保证了。例如,`f()`会把`Triangle*`插入其`set<Shape*>`参数里。如果这个`set<Shape*>`实际上是一个`set<Circle*>`,那也就摧毁了有关`set<Circle*>`的成员一定是`Circle`的基本保证。

13.6.3.1 模板转换

前一节的内容说明,由同一个模板生成的各个类之间并没有任何默认的关系。然而,对

① 由下面的例子可以看出,对于`Shape`的集合能够做的某些事情(例如插入一个三角形),对`Circle`的集合而言则是非法操作,`Shape`的集合不会遵守`Circle`的集合所要求的行为规范。——译者注

于某些模板，我们可能希望能表述这种关系。举个例子，假定我们定义了一个指针模板，我们就可能希望能反应出被指向的对象间的继承关系。成员模板（13.6.2节）使我们在需要的时候能够刻画这类关系。考虑

```
template<class T> class Ptr { // 到T的指针
    T* p;
public:
    Ptr(T*);
    Ptr(const Ptr&);           // 复制构造函数
    template<class T2> operator Ptr<T2> (); // 将Ptr<T> 转换到Ptr<T2>
    // ...
};
```

我们可能希望定义转换运算符，以便对这里用户定义的`Ptr`类型提供继承关系，这种关系是我们在内部指针类型中已经习以为常的。例如，

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;      // 应该能行
    Ptr<Circle> pc2 = ps;    // 应指出错误
}
```

当且仅当`Shape`是`Circle`的一个直接或者间接的公用基类时，我们就想允许前一个初始化。一般来说，我们需要定义一个转换运算符，使得当且仅当`T*` 能够赋值给`T2*` 时，从`Ptr<T>` 到`Ptr<T2>` 的转换能被接受。按如下方式就可以做到这一点：

```
template<class T>
    template<class T2>
        Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

当且仅当`p`（其类型是`T*`）可以作为`Ptr<T2>(T2*)` 构造函数的参数时，这个返回语句才能编译。因此，如果`T*` 可以隐式转换到`T2*`，从`Ptr<T>` 到`Ptr<T2>` 的转换就能工作。例如，

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;      // 可以：Circle* 能转换到Shape*
    Ptr<Circle> pc2 = ps;    // 错误：Shape* 不能转换到Circle*
}
```

应该小心，只去定义那些在逻辑上有意义的转换。

注意，一个模板的模板参数表与其模板成员的模板参数表不能组合在一起。例如，

```
template<class T, class T2> // 错误
    Ptr<T>::operator Ptr<T2> () { return Ptr<T2>(p); }
```

13.7 源代码组织

使用了模板的代码有两种明显的组织方式：

- [1] 在使用模板的编译单位最前面包含（include）有关模板的定义。
- [2] 只在使用模板的编译单位最前面包含（include）有关模板的声明，并且分别编译它们的模板定义。

此外，模板函数有时是首先声明，然后使用，最后在某个编译单位里定义。

要看看这两种方式之间的差异，考虑简单模板

```
#include<iostream>

template<class T> void out(const T& t) { std::cerr << t; }
```

我们可以将它命名为`out.c`，而后将它 `#include`到所有需要`out()`的地方。例如，

```
// user1.c:
#include "out.c"
// 使用out()

// user2.c:
#include "out.c"
// 使用out()
```

这样，`out()`的定义和它所依赖的所有声明都被 `#include`在几个不同的编译单位里。(仅)在需要时才生成代码，优化对冗余定义的读入过程等工作都是编译器的责任。

这种做法的一个明显问题是，`out()`的定义所依靠的所有东西都将被加入到每个使用`out()`的文件里，这就增大了编译器必须处理的信息量。另一个问题是，用户可能意外地依赖了原本只是为了定义`out()`的需要而包含进来的东西。通过使用名字空间，避免宏定义，以及一般性地减少包含进来的信息量等，可以使这种危险减到最小。

分别编译的策略是下面思想的逻辑结论：如果模板定义没有被包含到用户代码里，它所依赖的东西也就不会影响到用户代码了。为此，我们把`out.c`分成两个文件

```
// out.h:
template<class T> void out(const T& t);

// out.c:
#include<iostream>
#include "out.h"

export template<class T> void out(const T& t) { std::cerr << t; }
```

现在文件`out.c`保存着定义`out()`所需要的全部信息，而`out.h`里只保存了调用它所需要的信息。用户只 `#include`这个声明（界面）：

```
// user1.c:
#include "out.h"
// 使用out()

// user2.c:
#include "out.h"
// 使用out()
```

这种策略对模板函数的处理与对非在线函数的处理一样。`out()`定义（在`out.c`里）被单独进行编译，在需要这个定义时，找出它来是实现的责任。这种策略也给实现增加了负担：不是去过滤掉模板函数的冗余副本，而是在需要使用时，该实现必须能找到那个惟一的定义。

注意，为了使其他编译单位能够访问，就必须将这个模板定义显式地声明为`export`（9.2.3节），方式是为定义或者定义之前的某个声明加上关键字`export`。如果没有这样做，在任何地方要使用模板，都将要求它的定义必须位于作用域中。

哪种策略或者策略组合最好，依赖于所用的编译器和连接系统，依赖于你正在构造的应用的种类，以及对于你所构造的系统的外部约束。一般来说，那些基本上就是调用其他模板函数的在线函数和小模板函数，都是包含到使用它们的每个编译单位里的候选者。在那些连

接系统只能对模板实例化提供普通支持的实现里，采用这一做法可以加快编译速度，改善错误信息。

包含一个定义将使它容易受损，因为它的意义可能受到包含它的环境中的宏或者其他声明的影响。因此，对于大的模板函数和那些有着非平凡的环境依赖性的模板函数，最好采用分开编译。另外，如果一个模板的定义要求一大堆声明，将这些声明包含到模板的使用环境里，也有可能造成一些出乎意料的影响。

我认为，让模板函数的定义分开另行编译，在用户代码中只包含声明的方式是最理想的。然而，这个理想的使用也必须与实际约束相协调，在某些实现中分开编译模板的代价非常昂贵。

无论采用哪种策略，非`inline`的`static`成员（C.13.1节）都必须在某个编译单位里有惟一的定义。这也意味着，在那些需要包含到多个编译单位里去的模板里最好不要使用这种成员。

应该提供这样的一个理想：无论是作为一个单位编译，还是分成一些独立的编译单位，这些代码都应能以同样的方式工作。接近这种理想的方式应该是限制模板定义对其环境的依赖性，而不是在进入实例化过程时为它的定义带上尽可能多的环境信息。

13.8 忠告

- [1] 用模板描述需要使用到许多参数类型上去的算法；13.3节。
- [2] 用模板表述容器；13.2节。
- [3] 为指针的容器提供专门化，以减小代码规模；13.5节。
- [4] 总是在专门化之前声明模板的一般形式；13.5节。
- [5] 在专门化的使用之前先声明它；13.5节。
- [6] 尽量减少模板定义对于实例化环境的依赖性；13.2.5节、C.13.8节。
- [7] 定义你所声明的每一个专门化；13.5节。
- [8] 考虑一个模板是否需要有针对性C风格字符串和数组的专门化；13.5.2节。
- [9] 用表述策略的对象进行参数化；13.4节。
- [10] 用专门化和重载为同一概念的针对不同类型的实现提供统一界面；13.5节。
- [11] 为简单情况提供简单界面，用重载和默认参数去表述不常见的情况；13.5节、13.4节。
- [12] 在修改为通用模板之前，在具体实例上排除程序错误；13.2.1节。
- [13] 如果模板定义需要在其他编译单位里访问，请记住写`export`；13.7节。
- [14] 对大模板和带有非平凡环境依赖性的模板，应采用分开编译的方式；13.7节。
- [15] 用模板表示转换，但要非常小心地定义这些转换；13.6.3.1节。
- [16] 如果需要，用`constraint()`成员函数给模板的实参增加限制；13.9[16]，C.13.10节。
- [17] 通过显式实例化减少编译和连接时间；C.13.10节。
- [18] 如果运行时的效率非常重要，那么最好用模板而不是派生类；13.6.1节。
- [19] 如果增加各种变形而又不重新编译是很重要的，最好用派生类而不是模板；13.6.1节。
- [20] 如果无法定义公共的基类，最好用模板而不是派生类；13.6.1节。
- [21] 当有兼容性约束的内部类型和结构非常重要时，最好用模板而不是派生类；13.6.1节。

13.9 练习

1. (*2) 纠正13.2.5节`List`定义的错误，并请写出与编译器将对`List`和函数`f()`的定义生成的代码

等价的代码。用你手写的代码和编译器从模板版本生成的代码运行一些小的测试实例。如果你的系统提供了可能性的话，请对这些代码做一个比较。

2. (*3) 写出一个单链表类模板，它能接受由类**Link**派生出的任何类型的元素，**Link**类保存为链接元素所需要的信息。这种表称为侵入表。借助这个表写一个能接受任何类型的元素的单链表（非侵入表）。比较两种表类的性能，讨论在它们之间的各种平衡。
3. (*2.5) 写出侵入的和非侵入的双向链表。除了你认为必须为单链表提供的操作之外，还必须提供哪些操作？
4. (*2) 基于11.12节的**String**类完成13.2节的**String**模板。
5. (*2) 定义**sort()**，它以比较准则作为一个模板参数。定义包含两个数据成员**count**和**price**的类**Record**。根据各个数据成员对**vector<Record>**排序。
6. (*2) 实现**qsort()**模板。
7. (*2) 写一个程序读入 (**key**, **value**) 对，打印出对应每个不同**key**值的所有**value**值之和。说明对能够作为**key**或**value**的类型有哪些要求。
8. (*2.5) 基于11.8节的**Assoc**类实现一个简单的**Map**类。保证这个**Map**对于将C风格的字符串和**string**作为关键码都能正确工作。保证这个**Map**能够对没有默认构造函数的类型正确工作。为在**Map**的所有元素上进行迭代提供一种方式。
9. (*3) 对11.8节的单词统计程序和不使用关联数组的程序做一个比较。对两种情况使用同样风格的I/O。
10. (*3) 用更适当的数据结构（例如，红黑树或者斜树）重新实现13.9[8]的**Map**。
11. (*2.5) 用**Map**实现一个拓扑排序函数。拓扑排序在 [Knuth, 1968] 第一卷（第二版）第262页有介绍。
12. (*1.5) 使13.9[7]的求和程序对包含空格的名字（如“thumb tack”）也能正确工作。
13. (*2) 为不同形式的行写出模板**readline()**，例如 (**item**, **count**, **price**)。
14. (*2) 用13.4节所概述的技术对字符串按逆字典序排序。保证这种技术对于**char**是**signed**的C++实现和**char**是**unsigned**的C++实现都能工作。用这种技术的一种变形提供不区分大小写的排序。
15. (*1.5) 构造出一个例子，阐明在函数模板与宏之间至少有三种差异（不算它们在定义语法方面的差异）。
16. (*2) 设计出一种模式，保证编译器能够在构造对象的时候，对每个模板的模板参数检查某些一般性的约束条件。只检查形式为“参数**T**必须是由**My_base**派生出的一个类”的约束是不够的。

第14章 异常处理

我正被打断之时，
请别打断我。
——温斯顿·丘吉尔

错误处理——异常的结组——捕捉异常——捕捉所有异常——重新抛出——资源管理
——*auto_ptr*——异常和 *new*——资源耗尽——构造函数里的异常——析构函数里的异常
——非错误的异常——异常描述——未预期的异常——未捕捉的异常——异常和效率
——处理错误的其他方式——标准异常——忠告——练习

14.1 错误处理

正如8.3节所指出的，一个库的作者可以检查出运行时错误，但一般说却根本不知道怎样去处理它们。库的用户知道怎样对付这些错误，但却又无法去检查它们——要不然这些错误就会在用户的代码里处理了，不会留给库去发现。提供异常的概念就是为了有助于处理这类问题。在这里的基本想法是，让一个函数在发现了自己无法处理的错误时抛出（throw）一个异常，希望它的（直接或者间接）调用者能够处理这个问题。希望处理这类问题的函数可以表明它将要捕捉（catch）这个异常（2.4.2节、8.3节）。

与传统处理错误的技术相比，这种处理风格有许多优越性。请考虑其他的可能方式，在检查到一个在局部无法处理的问题时，一个函数可以：

- [1] 终止程序。
- [2] 返回一个表示“错误”的值。
- [3] 返回一个合法值，让程序处于某种非法的状态。
- [4] 调用一个预先准备好在出现“错误”的情况下用的函数。

情况 [1]，“终止程序”，也是在异常未被捕捉的情况下要默认发生的事情。对于大部分错误，我们都能够而且必须做得更好一些。特别是，一个库并不知道它所嵌入其中的程序的用途和一般性策略，绝不应该简单地调用 *exit()* 或者 *abort()*。无条件地终止程序的库将无法用到那些不能允许垮台的程序里。认识异常的一种方式，是将它看做在无法局部地执行有意义的动作时，就把控制交给调用者。

情况 [2]，“返回一个错误值”未必总可行，因为有时根本不存在可接受的“错误值”。举例来说，如果某个函数返回 *int*，每个 *int* 就都是可能出现的结果。即使在那些可以使用这种方式的地方，这样做起来也极不方便，因为每个调用都需要检查错误值。这很容易使程序的规模加倍（14.8节）。因此，这种方式极少能系统地使用以检查出所有的错误。

情况 [3]，“返回一个合法值，让程序处于某种非法的状态”也有问题，因为调用函数很可能没有注意到程序已经处在一种非法的状态中。例如，许多标准C库函数通过设置全局变量

`errno`指明出现了错误(20.4.1节、22.3节),然而,程序却常常没有足够经常地去检查`errno`,因此就不能避免由于失败的调用返回的值而引起后面的错误。进一步说,用全局变量记录错误的方式也不能在存在并发的情况下很好工作。

异常处理并不意味着要去处理那些与情况[4] (“调用一个错误处理函数”)有关系的问题。无论如何,如果缺乏异常这个概念,该错误处理函数在它如何处理错误时,能做出的选择正好就是另外那三种方式。有关错误处理函数和异常的进一步讨论,见14.4.5节。

异常处理机制是在传统技术不充分、不优美和容易出错的时候,提供的一种替代它们的技术。它提供了一种方法,能明确地把错误处理代码从“正常”代码中分离出来,这将使程序更容易读,也更容易用工具进行处理。异常处理机制提出了一种更规范的错误处理风格,这样就能简化分别写出的代码片段之间的相互关系。

异常处理模式有一个使C和Pascal程序员感到新颖的方面,它对于一个错误(特别是源自库的错误)的默认响应方式是终止程序。而在这时传统的反应则是装糊涂,接着做下去,以期能得到最好的结果。这样,异常处理机制将使程序更“脆弱”,这样说的意思是,为使程序的运行可以接受,就必须付出更多的关心和努力。与那种在开发阶段的后期——或甚至是在认为开发阶段已经结束,将程序递交给无辜的用户之后——得出的却是错误结果相比,这种情况看来是更受欢迎的。在那些无法被接受终止的地方,我们可以捕捉所有的异常(14.3.2节)或者捕捉某些种类的所有异常(14.6.2节)。这样,一个异常就能只在程序员希望终止的时候才会使程序终止。与传统方式下不完全的恢复机制导致了灾难性错误,以至出现无条件终止程序的情况相比,采用异常的方式当然更好些。

有时人试着通过写出一些错误信息,或者弹出一个对话框要求用户帮助等方式,来缓和“装糊涂”方式中最不吸引人的方面。在排除程序错误的阶段中,这种方式基本上是有用的,因为当时的用户就是程序员自己,他们熟悉程序的内部结构。到了非开发者的手里,一个库去向(可能并不存在的)用户/操作员寻求帮助,这样做根本就无法接受。另外,在许多情况下根本就没有放错误信息的地方(比如说,程序可能运行在某种环境里,其中的`cerr`没有连接到任何能使用户注意到的东西上);从哪个方面看它们也是最终用户无法理解的。至少,错误信息可能用的是某种不对路的自然语言(例如将芬兰语信息送给英语用户)。更糟的情况是,错误信息常常引证某些用户完全不知道的库概念(例如,由于对图形系统的错误输入产生“`atan2`参数错”)。好的库不会按照这种方式“饶舌”。异常为写代码提供了一种方式,使它们可以在检查出自己无法恢复的问题时,将此问题传递给系统中的某个可能完成恢复工作的部分。也只有那种对于程序运行的环境有所了解的系统部分,才有可能组合出一条有意义的错误信息。

异常处理机制可以看做是编译时的类型检查和歧义性控制机制在运行中的对应物。它也使设计过程变得更加重要。为了使一个程序初始的遍布着错误的版本得以运行,需要做的工作有可能会增加。但是,这样做的结果是,代码有更大的可能性能够如预期的那样运行,能够作为更大的程序中的一部分运行,更容易为其他程序员所理解,更容易用各种工具去操作。与此类似,异常处理所提供的这些特殊语言特征还支持一种“好的风格”,其方式与C++语言的其他特征支持“好的风格”一样。而在如C或者Pascal那样的语言里,这种好风格只能非形式地、不完全地去实践。

应该认识到,错误的处理将仍然是一件很困难的工作。与那些只涉及局部控制流的语言

特征相比，异常处理机制——虽然远比它要取代的那些技术更规范——相对而言仍然是最无结构性的。C++ 的异常处理机制为程序员提供了一种处理错误的方式，使他们能在给定的系统结构中，能以最自然的方式去处理这些错误。异常机制也使处理错误的复杂性更加清晰可见。当然不是异常导致了那些复杂性，请当心，不要因为听到坏消息就去骂报信的人。

现在是重温8.3节的好时候，在那里介绍了异常处理的基本语法、语义和使用风格等诸方面的情况。

14.1.1 关于异常的其他观点

“异常”也是这样的—一个词，对于不同的人它常常意味着不同的东西。C++ 异常处理机制的设计是为了处理错误和其他非正常的情况（并由此得名）。应指出，这里特别希望它能够支持由分别开发的组件组合而成的程序中的错误处理。

这个机制的设计只是为了处理同步异常，例如数组范围检查和I/O出错。异步事件，如键盘中断和明确的算术错误不一定是异常，也不能直接由这种机制处理。要能清晰而高效地处理异步事件，需要某种从本质上与异常（按照这里的定义）不同的机制。许多系统提供了用于处理异步问题的有关机制（如信号）。由于那些东西依赖于具体的系统，因此在这里将不予描述。

异常处理机制是一种非局部的控制结构，基于堆栈回退（stack unwinding, 14.4节），因此也可以看做是另一种返回机制。存在着许多并不是处理错误的对异常的合法使用（14.5节）。然而，异常处理机制的基本目标和本章的关注点仍然是错误处理和对容错的支持。

标准C++ 里没有线程或者进程的概念。因此，这里将不讨论与并发有关的异常环境问题。在你所用系统里，并发功能应该在其文档中有所描述。在这里，我只想提一下，C++ 中异常处理机制的设计能够有效用于并发程序，只要程序员（或者系统）遵守基本的并发规则，例如在使用共享资源期间正确地将其锁定。

C++ 的异常处理机制能够用于报告和错误与异常事件。然而，程序员必须决定在一个给定的程序，应该将哪些情况当做异常。这件事并不总是很容易做的（14.5节）。在一个程序运行的大部分时间都可能发生的一个事件能否当做一个异常？一个在计划中的并能予以处理的事件能否作为一个错误？对于两个问题的回答都是肯定的。“异常”并不意味着“几乎不出现”或者“灾难性的”。最好是把异常想像为表示“系统的某些部分不能完成要它做的事情了”。通常我们可以在此之后试着去做些其他的事情。异常的`throw`应该没有函数调用那么频繁，否则这个系统的结构就会变模糊。当然，我们应该期望，大部分大型程序，在其正常的成功运行中，也至少会`throw`和`catch`其中的某些异常。

14.2 异常的结组

一个异常也就是某个用于表示异常发生的类的一个对象。检查到一个错误的代码段（常常是某个库）`throw`一个对象（8.3节）。一个代码段用`catch`子句表明它要处理的某个异常。一个`throw`的作用就是导致堆栈的一系列回退，直到找到某个适当的`catch`（在某个直接或间接地调用了抛出异常的那个函数的函数里）。

异常经常可以自然地形成一些族。这就意味着可以借助于继承来表示异常的结构，以帮助异常处理。例如，某个数学库的异常可以用：

```

class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...

```

方式组织，这就使我们可以处理所有的**Matherr**，而不必去考究它到底属于哪一类。例如，

```

void f()
{
    try {
        // ...
    }
    catch (Overflow) {
        // 处理Overflow或者任何由Overflow派生的异常
    }
    catch (Matherr) {
        // 处理所有不是Overflow的Matherr
    }
}

```

在这段程序里，**Overflow**将得到特殊处理，而所有其他**Matherr**都作为一般的情况处理。

将异常组织为一些层次结构对于代码的健壮性可能很重要。例如，考虑一下如果没有这种结组机制，你将如何处理来自一个数学库里的所有异常。这当然可以通过完全地列举出这些异常的方式做到：

```

void g()
{
    try {
        // ...
    }
    catch (Overflow) { /* ... */ }
    catch (Underflow) { /* ... */ }
    catch (Zerodivide) { /* ... */ }
}

```

这不仅会令人厌烦，而且程序员也会不经意地忘记把某个异常列入表中。再来考虑如果我们没有将数学异常结组会出现什么情况。当我们给数学库加进一个新异常时，每一段试图处理所有数学异常的代码都必须修改。一般来说，在某个库的初始发布之后，这种广泛性的更新就已经不可行了。常常无法找到与此有关的所有代码段。即使可以找到，我们也无法一般性地假定所有源代码片段都可用，或即使都有我们也不愿意去修改它们。这些重新编译和维护问题将导致一种策略，那就是在库的第一个发布之后就不能再加入新的异常了；而这种策略对于几乎所有的库都是无法接受的。这种推理的结果是，异常经常被按照一个个库或者按照一个个子系统，定义为一些层次结构（14.6.2节）。

请注意，不论是内部的数学操作，还是基本的数学库（与C共享），都没有将算术错误报告为异常。出现这种情况的原因是，检查某些算术错误，例如除0，在许多流水线机器系统结构中都是非同步的操作。这里所描述的**Matherr**只是一个说明。各种标准库异常将在14.10节描述。

14.2.1 派生的异常

对异常处理采用类层次结构，将会很自然地产生一些异常处理器，它们只对异常所携带

信息中的一个子集感兴趣。换句话说，捕捉到某个异常的经常是针对其基类的处理器，而不是正好针对这个异常本身的类的处理器。捕捉和命名异常的语义等同于函数接受参数的语义。也就是说，用实际参数的值对形式参数做初始化（7.2节）。这也隐含着抛出的异常可能因为捕捉而被“切割”（12.2.3节）。例如，

```
class Matherr {
    // ...
    virtual void debug_print() const { cerr << "Math error"; }
};

class Int_overflow: public Matherr {
    const char* op;
    int a1, a2;
public:
    Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
    virtual void debug_print() const { cerr << op << ' (' << a1 << ', ' << a2 << ' ) '; }
    // ...
};

void f()
{
    try {
        g();
    }
    catch (Matherr m) {
        // ...
    }
}
```

在进入`Matherr`处理器之后，`m`是一个`Matherr`对象——即使`g()`的调用抛出的实际是`Int_overflow`。这也意味着`Int_overflow`所携带的附加信息将是不可访问的。

如常，这里也可以用指针或引用来避免信息的永久性丢失。例如我们可以写：

```
int add(int x, int y)
{
    if ( (x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y) )
        throw Int_overflow("+", x, y);

    return x+y;    // x+y没有溢出
}

void f()
{
    try {
        int i1 = add(1, 2);
        int i2 = add(INT_MAX, -2);
        int i3 = add(INT_MAX, 2);    // 这里发生异常
    }
    catch (Matherr& m) {
        // ...
        m.debug_print();
    }
}
```

在这里，`add()`的最后调用将触发一个异常，并导致对`Int_overflow::debug_print()`的调用。如果这个异常是通过值调用而不是通过引用调用，那么被调用的就会是`Matherr::debug_print()`。

14.2.2 多个异常的组合

并不是每组异常都具有树形结构。也常有一个异常同属于两个组的情况。例如，

```
class Netfile_err : public Network_err, public File_system_err { /* ... */};
```

这样的—个*Netfile_err*异常能够被处理网络异常的函数捕捉

```
void f()
{
    try {
        // 某些事情
    }
    catch (Network_err& e) {
        // ...
    }
}
```

也能被处理文件系统异常的函数捕捉

```
void g()
{
    try {
        // 某些其他事情
    }
    catch (File_system_err& e) {
        // ...
    }
}
```

对于错误处理的这种非层次结构性的组织形式也很重要，特别是当有关的服务——例如连网——对用户完全透明时。在这种情况下，写*g()*的人甚至根本就没有意识到还牵涉到网络（14.6节）。

14.3 捕捉异常

考虑

```
void f()
{
    try {
        throw E();
    }
    catch (H) {
        // 何时我们能到这里？
    }
}
```

执行将会涉及这个处理器，

- [1] 如果*H*是与*E*相同的类型。
- [2] 如果*H*是*E*的无歧义的公用基类。
- [3] 如果*H*和*E*是指针类型，且 [1] 或 [2] 对它们所引用的类型成立。
- [4] 如果*H*是引用类型，且 [1] 或 [2] 对*H*所引用的类型成立。

此外，我们还可以给用于捕捉异常的类型加上*const*，就像我们可以给函数参数加上*const*—样。这不会改变能捕捉到的异常集合，而只是限制我们，不能去修改捕捉到的那个异常。

从原则上说,异常在抛出时将被复制,所以,处理器得到的只是原始异常的一个副本。事实上,一个异常有可能在被捕捉之前复制过许多次。因此,我们不应该抛出一个不允许复制的异常。具体实现可以应用各种各样的策略去存储和传播异常。但无论如何,有一点是有保证的,系统中总存在着足够的存储,使`new`可以抛出一个标准的存储耗尽异常`bad_alloc`(14.4.5节)。

14.3.1 重新抛出

在捕捉了一个异常之后,处理器也常常会确定自己没办法完成对这个错误的全部处理。在这种情况下,典型的方式是该处理器做完局部能够做的事情,然后再一次抛出这个异常。这样就使错误能在最合适的地方被处理。如果在任何单独的地方都无法获得所需的全部信息,使错误得到最好的处理,那么最好是按这种方式,最好将恢复动作分布在几个处理器里。例如,

```
void h()
{
    try {
        // 可能抛出Matherr错误的代码
    }
    catch (Matherr) {
        if (can_handle_it_completely) {
            // 处理Matherr
            return;
        }
        else {
            // 完成在这里能做的事情
            throw;    // 重新抛出异常
        }
    }
}
```

重新抛出采用一个不带运算对象的`throw`表示。如果企图重新抛出,而这时又没有异常可以重新抛出,那么就会调用`terminate()`(14.7节)。编译器能够检查出这类情况中的一些并做出警告,但无法全部查出。

重新抛出的就是原来捕捉到的那个异常,而不是它的作为`Matherr`所能访问的那个部分。换句话说,如果原来抛出的是`Int_overflow`,调用`h()`的函数仍然能捕捉一个`Int_overflow`,它也就是被`h()`作为`Matherr`捕捉而后又决定重新抛出的那个异常。

14.3.2 捕捉所有异常

这一“捕捉并重新抛出”技术的某种退化版本也非常重要。对于函数,省略号`...`表示“任何参数”(7.6节),同样,`catch(...)`表示要“捕捉所有异常”。例如,

```
void m()
{
    try {
        // 一些代码
    }
    catch (...) {    // 处理所有异常
        // 清理
    }
}
```

```

        throw;
    }
}

```

在这里，如果在执行`m()`主要部分的结果是出现了任何异常，在处理器中的清理工作就会执行。一旦局部清理完成，导致这个清理的异常又会被重新抛出，去激发进一步的错误处理工作。查看14.6.3.2节所提出的一种技术，可以通过该技术去获取由...处理器捕捉到的异常里的信息。

一般意义下的错误处理，特别是异常处理，都需要维持程序所假定的某一种不变关系（24.3.7.1节）。例如，如果`m()`被假定要在它找到某些指针的那个状态中保留下这些指针，那么我们就可以在处理器里写代码，将可接受的值赋给这些指针。这样，利用“捕捉任何异常”的处理器就可以用于维护任意的不变关系。然而，对于许多重要的实际情况，这样一个处理器并不是解决问题的最优美的方法（14.4节）。

14.3.2.1 处理器的顺序

由于派生异常可能被多于一个异常类型的处理器捕捉，在与`try`语句时处理器的排列顺序就很重要了。处理器将被顺序检查。例如，

```

void f()
{
    try {
        // ...
    }
    catch (std::ios_base::failure) {
        // 处理任何流io错误（14.10节）
    }
    catch (std::exception& e) {
        // 处理所有标准库异常（14.10节）
    }
    catch (...) {
        // 处理任何其他异常（14.3.2节）
    }
}

```

因为编译器知道类层次结构，所以它能捕捉到许多逻辑错误。例如，

```

void g()
{
    try {
        // ...
    }
    catch (...) {
        // 处理任何异常（14.3.2节）
    }
    catch (std::exception& e) {
        // 处理所有标准库异常（14.10节）
    }
    catch (std::bad_cast) {
        // 处理dynamic_cast失败（15.4.2节）
    }
}

```

在这里的`exception`绝不会被考虑。即使我们删去了“捕捉一切”的处理器，`bad_cast`也不会

被考虑，因为它是由`exception`派生出来的。

14.4 资源管理

当一个函数申请了某种资源——譬如说，打开了一个文件，在自由空间分配了一些存储，设置过某种访问控制锁等——为了保证系统将来的运行，一个至关重要的问题就是应该正确地释放这些资源。通常所说的做到了“正确释放”，也就是要求申请资源的函数在返回其调用者之前完成这种释放。例如，

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r");
    // 使用f
    fclose(f);
}
```

这看起来像是正确的，直到你意识到，如果有什么事情在调用`fopen()`之后，调用`fclose()`之前出了毛病，一个异常就会导致`use_file()`退出，但却没有去调用`fclose()`。不支持异常的语言里也会发生同样的问题。例如，标准C库函数`longjmp()`就会导致同一问题。即使是一个常规的`return`语句也会使`use_file()`退出而没有关闭`f`。

设法使`use_file()`能够容错的第一个努力可能具有下面的样子：

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r");
    try {
        // 使用f
    }
    catch (...) {
        fclose(f);
        throw;
    }
    fclose(f);
}
```

将使用文件的代码包裹在一个`try`块里，让这个块捕捉所有异常，关闭文件并重新抛出这个异常。

这种解决方法的缺点是太罗嗦，令人厌倦，而且很可能代价高昂。进一步说，任何罗嗦并令人厌倦的解决方法都很容易出错，由于程序员会感到很讨厌。幸运的是，存在着一种更优雅的方法。问题的一般形式看起来具有下面的样子：

```
void acquire()
{
    // 申请资源1
    // ...
    // 申请资源n

    // 使用资源

    // 释放资源n
    // ...
    // 释放资源1
}
```


按照资源被申请的相反顺序将它们释放常常也很重要。这非常像局部对象由构造函数创建，并由析构函数销毁的情况。所以，我们只要适当地利用带有构造函数和析构函数的类的对象，就可以处理这种资源申请和释放问题。比如说，我们可以定义一个类 `File_ptr`，它就像是 `FILE*`：

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a) { p = fopen(n, a); }
    File_ptr(FILE* pp) { p = pp; }
    // 适当的复制操作
    ~File_ptr() { if (p) fclose(p); }

    operator FILE*() { return p; }
};
```

在给了一个 `FILE*` 或者 `fopen()` 所需要的参数，我们就可以创建起一个 `File_ptr`。在这两种情况下，这个 `File_ptr` 在其作用域结束时将被销毁，它的析构函数就会关闭对应的文件。现在我们的函数缩到了最短的程度：

```
void use_file(const char* fn)
{
    File_ptr f(fn, "r");
    // 使用f
}
```

那个析构函数总会被调用，无论该函数是正常结束，还是因为抛出了某个异常而退出。在这里，异常处理机制使我们能够从主要算法中删去处理错误的代码[⊖]，这样得到的代码更简单，与其传统对应物相比也更不容易出错。

“向上穿过堆栈”去为某个异常查找对应处理器的过程，通常被称为“堆栈回退”。在堆栈回退的过程中，将会对所有构造起来的局部对象调用析构函数。

14.4.1 构造函数和析构函数的使用

利用局部对象管理资源的技术通常被说成是“资源申请即初始化”。这种技术依赖于构造函数和析构函数的性质，以及它们与异常处理的相互关系。

只有在一个对象的构造函数执行完成时，这个对象才被看做已经建立起来了。此后，也只有在此之后，堆栈回退时才为该对象调用析构函数。一个由子对象组成的对象的构造将一直持续到它所有的子对象都完成了构造工作。数组的构造一直持续到它的所有元素都构造完成（而且，只有那些构造完成的元素才会在堆栈回退时销毁）。

构造函数试图去保证对象能够完全、正确地创建起来。如果这个目标无法达到，那么书写良好的构造函数就应当将系统的状态（尽可能地）恢复到对象开始构造之前的情况。理想状况是，按朴素方式写出的构造函数总能达到各种可能情况之中的一种，而不会使它们处理

⊖ 这个说法不准确，异常处理只是这一解决方案的基础的一部分。构造函数/析构函数和局部作用域规则使这种技术可行：局部对象将在作用域结束时销毁，如果它有析构函数，销毁时就会自动调用有关析构函数。当然，还必须保证程序中没有用那“不负责任”的非局部退出机制——那种能退出局部作用域，而又不正常销毁局部变量的机制，如上面所述的 `longjmp` 之类。——译者注

的对象处于某种“部分构造好了”状态。通过对成员应用“资源申请即初始化”技术，就可以做到这一点。

考虑类 X ，它的构造函数需要申请两种资源：文件 x 和锁 y 。这些请求都可能失败并抛出异常。类 X 的构造函数绝不应该在得到了文件但未得到锁的情况下返回。进一步说，也不应该通过给程序员添加很大负担的方式来满足这种要求。我们使用两个类 $File_ptr$ 和 $Lock_ptr$ 表示所申请的资源。对某项资源的申请用对代表该资源的对象的初始化来表示：

```
class X {
    File_ptr aa;
    Lock_ptr bb;
public:
    X(const char* x, const char* y)
        : aa(x, "rw"), // 申请'x'
          bb(y)         // 申请'y'
    {}
    // ...
};
```

现在就像局部变量的情况一样，这个实现将能完成所有的簿记工作，而用户则完全不必去追踪任何情况。例如，如果在创建了 aa 但还没有 bb 的情况下出现了异常， aa 的析构函数就会被调用， bb 的则不会。

这也意味着，在所有遵循这种资源申请的简单模型的地方，写构造函数的人就不必专门去写显式的异常处理代码。

以某种特定方式申请的最普通的资源就是存储。例如，

```
class Y {
    int* p;
    void init();
public:
    Y(int s) { p = new int[s]; init(); }
    ~Y() { delete[] p; }
    // ...
};
```

这种方式在实际中常常能看到，它有可能导致“存储流失”。如果 $init()$ 抛出异常，那么申请到的存储就不会被释放，因为有关对象并没有构造完成，对它不会调用析构函数。一种安全的变形是

```
class Z {
    vector<int> p;
    void init();
public:
    Z(int s) : p(s) { init(); }
    // ...
};
```

现在，通过 p 使用的资源是由 $vector$ 管理的。如果 $init()$ 抛出异常，在 p 的析构函数被调用时，已经申请到的存储就会被释放。

14.4.2 auto_ptr

标准库提供了模板类 $auto_ptr$ ，支持“资源申请即初始化”的技术。简而言之， $auto_ptr$

可以用指针去初始化，且能以与指针同样的方式间接访问。还有就是，在`auto_ptr`退出作用域时，被它所指的对象将被隐式地自动删除。例如，

```
void f(Point p1, Point p2, auto_ptr<Circle> pc, Shape* pb) // 记住，退出时需删除pb
{
    auto_ptr<Shape> p(new Rectangle(p1, p2)); // p指向一个矩形
    auto_ptr<Shape> pbox(pb);

    p->rotate(45); // auto_ptr<Shape> 的使用与Shape* 一样
    // ...
    if (in_a_mess) throw Mess();
    // ...
}
```

这里的`Rectangle`（由`pb`所指的`Shape`）和由`pc`所指的`Circle`都将被删除，无论是否有异常被抛出。

为获得这种所有权语义（ownership semantics），也常被称做破坏性复制语义（destructive copy semantics），`auto_ptr`具有与常规指针很不一样的复制语义：在将一个`auto_ptr`复制给另一个之后，原来的`auto_ptr`将不再指向任何东西。因为复制`auto_ptr`将导致对它本身的修改，所以`const auto_ptr`就不能复制。

`auto_ptr`模板在 `<memory>` 里声明，它可以用下面的实现描述：

```
template<class X> class std::auto_ptr {
    template<class Y> struct auto_ptr_ref { /* ... */ }; // 协助类
    X* ptr;
public:
    typedef X element_type;

    explicit auto_ptr(X* p=0) throw() { ptr=p; } // throw() 说明“不抛出异常”，见14.6节
    ~auto_ptr() throw() { delete ptr; }

    // 注意，复制构造函数和赋值都用非const参数
    auto_ptr(auto_ptr& a) throw(); // 复制，而后a.ptr = 0
    template<class Y> auto_ptr(auto_ptr<Y>& a) throw(); // 复制，而后a.ptr = 0
    auto_ptr& operator=(auto_ptr& a) throw(); // 复制，而后a.ptr = 0
    template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw(); // 复制，而后a.ptr = 0

    X& operator*() const throw() { return *ptr; }
    X* operator->() const throw() { return ptr; }
    X* get() const throw() { return ptr; } // 提取指针
    X* release() throw() { X* t = ptr; ptr=0; return t; } // 放弃所有权
    void reset(X* p=0) throw() { if (p!=ptr) { delete ptr; ptr=p; } }

    auto_ptr(auto_ptr_ref<X>) throw(); // 从auto_ptr_ref复制
    template<class Y> operator auto_ptr_ref<Y>() throw(); // 复制到auto_ptr_ref
    template<class Y> operator auto_ptr<Y>() throw(); // 从auto_ptr的破坏性复制
};
```

`auto_ptr_ref`的用途就是为普通`auto_ptr`实现破坏性复制语义，这也使`const auto_ptr`不可能复制。如果指针`D*`能转换到`B*`，那么这里的模板构造函数和模板赋值都能（显式或隐式地）将`auto_ptr<D>`转换到`auto_ptr`。例如，

```
void g(Circle* pc)
{
    auto_ptr<Circle> p2(pc); // 现在p2负责删除
    auto_ptr<Circle> p3(p2); // 现在p3负责删除（且p2不再负责）
}
```

```

    p2->m = 7;           // 程序错误: p2.get() == 0
    Shape* ps = p3.get(); // 从auto_ptr抽取指针
    auto_ptr<Shape> aps(p3); // 转移所有权, 并转换类型
    auto_ptr<Circle> p4(pc); // 程序错误: 现在p4也负责删除
}

```

让多个`auto_ptr`拥有同一个对象的效果是无定义的; 最可能的情况是使该对象被删除两次(因而造成恶劣结果)。

注意, `auto_ptr`的破坏性复制语义也意味着它不满足标准容器或标准算法(例如, `sort()`)对元素的基本要求。例如,

```

vector< auto_ptr<Shape> > &v; // 危险: 在容器里使用auto_ptr
// ...
sort(v.begin(), v.end());      // 别这么做: 这个排序可能造成v混乱

```

显然, `auto_ptr`并不是一种通用的灵巧指针。然而, 它也确实提供了其设计所希望提供的服务——自动指针的异常安全性, 而且完全没有额外开销。

14.4.3 告诫

并不是所有的程序都需要有从所有破坏中恢复起来的能力。并不是所有资源都重要到需要付诸努力, 用“资源申请即初始化”技术、`auto_ptr`和`catch(...)`等去保护它们。例如, 对于许多只是简单地得到一些输入就一直运行到结束的程序, 如果在运行时发生了严重错误, 最合适的响应方式就是使这个进程夭折(在产生一些适当的诊断信息之后)。也就是说, 让系统去释放程序所申请的所有资源, 让用户重新用更合适的输入去运行程序。这里讨论的策略是想用在某些应用中, 对于它们, 对运行时错误的这种过于简单化的响应是无法接受的。特别是库的设计者, 他们通常不能在容错需求方面对使用这个库的程序做出任何假设, 因此就必须避免所有无条件的运行时失败, 而且必须要求库函数在返回调用程序前释放所有的资源。“资源申请即初始化”策略, 结合用异常去发出失败信号等, 对于许多这样的库十分合适。

14.4.4 异常和new

考虑

```

void f(Arena& a, X* buffer)
{
    X* p1 = new X;
    X* p2 = new X[10];
    X* p3 = new (&buffer[10]) X;           // 将X放入buffer(无需分配存储)
    X* p4 = new (&buffer[11]) X[10];
    X* p5 = new(a) X;                     // 从分配场a分配存储(从a中释放)⊖
    X* p6 = new(a) X[10];
}

```

如果`X`的构造函数抛出异常, 那又会怎样? 通过`operator new()`分配的存储能释放吗? 在普通的情况下对此的回答是“能”, 所以对`p1`和`p2`的初始化不会导致存储的流失。

[⊖] 这里的`a`是一个分配场(10.4.11节)。从`a`中分配一个`X`并将其交给`p5`, 从分配场`a`的角度看, 就像是它释放了一块存储。因此作者采用了这个说法。——译者注

如果采用的是放置语法 (10.4.11节), 问题的回答就不那么简单了。这种语法的一些使用实际分配了存储, 而后当然应该释放; 然而, 也有一些使用并没有分配存储。进一步说, 在使用放置语法的那些地方是想做某种非标准的分配, 所以经常需要做非标准的释放。因此采用何种行为依赖于使用的分配器。如果用的是分配器 `Z::operator new()`, 如果 `Z::operator delete()` 存在, 那么就应该调用它; 否则就不必去释放。数组也应该同样处理 (15.6.1节)。这一策略正确地处理了标准库的放置式 `new` 运算符 (10.4.11节)。如果程序员提供了成对的相互匹配的分配和释放函数, 采用这种策略也都能正确处理。

14.4.5 资源耗尽

对某种资源的申请失败时应该做什么, 这是程序设计中反复出现一个问题。例如, 在此之前我们轻松地打开了文件 (用 `fopen()`) 或者从自由空间申请了内存 (用运算符 `new`), 根本没有留心如果这个文件不存在或者已经用光了所有自由存储时会发生什么事情。在面对这种问题时, 程序员提出了两种风格的解决方案:

唤醒: 请求某个调用程序纠正问题, 而后继续执行。

终止: 结束当前计算并返回某个调用程序。

按照前一种方式, 一个调用程序必须准备好, 去帮助处理某段未知的代码中出现的资源申请问题。对于后者, 一个调用程序必须准备好去应付某个资源申请失败的情况。对于大部分情况, 做到后者都远为简单, 也使系统能保持抽象层次之间的一种较好的隔离。请注意, 在采取终止策略时, 终止的并不是整个程序, 而只是其中某个个别的计算。“终止”是一个传统的描述策略的术语, 表示的是从“失败的”计算返回到与某个调用过程相关的某个错误处理器 (它当然也可以重试那个失败的计算), 而不是试图去修复那个坏状况, 而后从检查出问题的那一点继续下去。

在C++里, 唤醒模型由函数调用机制支持, 而终止模型由异常处理机制支持。这两种情况可以利用标准库 `operator new()` 的一个简单实现及其使用阐释清楚:

```
void* operator new(size_t size)
{
    for (;;) {
        if (void* p = malloc(size)) return p;           // 试图找到存储
        if (_new_handler == 0) throw bad_alloc();       // 没有处理器; 放弃
        _new_handler();                                 // 寻求帮助
    }
}
```

在这里, 我借助于标准C库 `malloc()` 完成对存储的实际检索工作, `operator new()` 的其他实现也可能选择其他方式。如果找到存储, `operator new()` 就能返回指向该存储的指针。如果找不到, `operator new()` 就调用 `_new_handler`。如果 `_new_handler` 能找到更多的存储供 `malloc()` 去分配, 那就万事大吉了。如果它也找不到, 那么它不可能返回 `operator new()` 而又不导致无穷循环。所以这时 `_new_handler()` 就可能选择抛出一个异常, 把这个麻烦留给某个调用者去处理

```
void my_new_handler()
{
    int no_of_bytes_found = find_some_memory();
}
```

```
    if (no_of_bytes_found < min_allocation) throw bad_alloc();    // 放弃
}
```

在某个地方，应该有另一个带有适当处理器的try块

```
try {
    // ...
}
catch (bad_alloc) {
    // 对存储耗尽的某种回应
}
```

在operator new()的实现中所用的_new_handler是一个指向函数的指针，由标准函数set_new_handler()维护。如果我希望将my_new_handler()作为_new_handler使用，就写

```
set_new_handler(&my_new_handler);
```

如果我还想捕捉bad_alloc，那么就可以写

```
void f()
{
    void (*oldnh) () = set_new_handler(&my_new_handler);

    try {
        // ...
    }
    catch (bad_alloc) {
        // ...
    }
    catch (...) {
        set_new_handler(oldnh); // 重置处理器
        throw;                // 重新抛出
    }

    set_new_handler(oldnh);    // 重置处理器
}
```

还有更好的方法，也可以对_new_handler应用14.4节描述的“资源申请即初始化”技术(14.12[1])，以避免用catch(...)处理器。

这个_new_handler并没有将更多信息从查出错误的地方传递到能起帮助作用的函数。要传递更多的信息也不难。但是，从检查运行时错误的代码向帮助处理这个错误的函数传递的信息越多，这两段代码之间的相互依赖性也就越强。这也意味着，对于两段代码中的某一段的修改将要求理解另一段代码，甚至要求去修改它。为保持软件中分离的代码相互隔离，尽可能地减少这种依赖性是很好的想法。与去调用由某调用函数提供的一个帮助例程相比，异常处理机制对这种隔离的支持更好一些。

一般来说，将资源分配组织在一些层次中（抽象的层次中），避免让一层依赖于调用它的另一层提供的帮助，这样做是很明智的。对大型系统的经验显示，许多成功的系统都在向这个方向演化。

要想抛出一个异常，就要求存在一个能抛出的对象。每个C++实现都要求保留足够的存储，在存储耗尽的情况下还能抛出bad_alloc。然而，由于抛出其他对象而导致存储耗尽的情况也是可能发生的。

14.4.6 构造函数里的异常

异常也为从构造函数里报告出错的问题提供了一个解决方案。由于构造函数无法返回一个独立的值供调用程序检查，传统的（即非异常处理的）可能的解决办法有：

- [1] 返回一个处于错误状态的对象，相信用户有办法检查其状态。
- [2] 设置一个非局部变量（例如，*errno*）指出创建失败，相信用户能去检查这个变量。
- [3] 在构造函数中不做初始化，依靠用户在第一次使用对象之前调用某个初始化函数（E.3.5节）。
- [4] 将对象标记为“未初始化的”，让对这个对象调用的第一个成员函数去完成实际的初始化工作，并让这个函数在初始化失败时报告错误。

异常处理机制使构造失败的信息能从构造函数内部传出来。例如，一个简单的**Vector**类可能在遇到超量存储要求时采用如下方式保护自己：

```
class Vector {
public:
    class Size { };
    enum { max = 32000 };
    Vector(int sz)
    {
        if (sz<0 || max<sz) throw Size();
        // ...
    }
    // ...
};
```

创建**Vector**的代码现在可以捕捉**Vecot::Size**错误，我们可以试着对此做些有意义的事情：

```
Vector* f(int i)
{
    try {
        Vector* p = new Vector(i);
        // ...
        return p;
    }
    catch (Vector::Size) {
        // 处理size错误
    }
}
```

像其他地方一样，错误处理器本身也可以使用标准的基本技术集合来处理错误报告和恢复的问题。每当异常被传到一个调用函数，有关什么东西出了错的观点也改变了。如果随异常传递过来适当的信息，为处理问题所能使用的信息也可能会增加。换句话说，错误处理技术的基本目标就是传递有关错误的信息，从检查的那一点传到存在着足够的可用信息，可以从错误中恢复出来的那一点，并能可靠而方便地这样做。

如果要处理那些要求多种资源的构造函数，“资源申请即初始化”技术是最安全最优美的方法（14.4节）。从根本上说，这种技术把处理多种资源的问题，归结为一种反复应用处理单一资源的（简单）技术的问题。

14.4.6.1 异常和成员初始化

如果一个成员的初始式（直接或者间接）抛出异常，那么会出现什么问题呢？按照默认约定，这个异常将传到调用这个成员的类的构造函数的位置。不过，构造函数本身也可以通过将完整的函数体——包括成员初始式表——包在一个`try`块里，自己设法捕捉这种异常。例如，

```
class X {
    Vector v;
    // ...
public:
    X(int);
    // ...
};

X::X(int s)
try
    : v(s)    // 用s初始化v
{
    // ...
}
catch (Vector::Size) { // 初始化v抛出的异常在这里捕捉
    // ...
}
```

14.4.6.2 异常和复制

与其他构造函数一样，复制构造函数也可以通过抛出异常的方式发出一个失败信号。在这种情况下，实际上并没有创建对象。例如，`vector`的复制构造函数常常需要分配存储并复制元素（16.3.4节、E.3.2节），因此可能导致异常的抛出。在抛出异常之前，复制构造函数就需要释放它已经申请到的所有资源。请查看E.2节和E.3节中有关容器类异常处理和资源管理的详尽讨论。

复制赋值在这方面与复制构造函数类似，它也可能申请资源，也可能通过抛出异常而结束。在抛出异常之前，赋值也必须保证它的每个操作对象都保持在某种合法状态。否则就可能了违背标准库的基本要求，从而导致无定义的行为（E.2节、E.3.3节）。

14.4.7 析构函数里的异常

从异常处理的角度看，析构函数可能在两种情况下被调用：

- [1] 正常调用：作为某个作用域正常退出的结果（10.4.3节），作为一个`delete`操作（10.4.5节）的结果等。
- [2] 在异常处理中被调用：在堆栈回退过程中（14.4节），异常处理机制退出一个作用域，其中包含有析构函数的对象。

对于后一种情况，绝不能让析构函数里抛出异常。如果真是这样，那就将被认为是异常处理机制的一次失败，并调用`std::terminate()`（14.7节）。归根到底，无论是异常处理机制，或者是这个析构函数，都没有一种普遍有效的方式来确定应该偏向哪个异常，忽略哪个异常^①。通过抛出异常退出析构函数也违背了标准库的基本要求（E.2节）。

① 请注意，这时有两个异常需要考虑：一个是导致该析构函数被调用的异常，另一个是执行析构函数的过程中发生的异常。——译者注

如果某个析构函数要调用一个可能抛出异常的函数，它可以保护自己，例如，

```
X::~~X()
try {
    f(); // 可能抛出
}
catch (...) {
    // 做某些事
}
```

如果有某个异常已经被抛出，但尚未被捕捉，标准库函数`uncaught_exception()`就会返回`true`。这就使程序员能在析构函数里，根据对象是被正常销毁还是作为堆栈回退中的一部分，描述不同的动作。

14.5 不是错误的异常

如果某个异常是有预期的，而且被捕捉到，所以根本不会对程序的行为产生不良影响，那怎么还能说它是错误呢？只不过是认为程序员认为它是一个错误，而且认为异常处理机制就是处理错误的工具。换一种看法，我们也可以认为异常处理机制不过是另一种控制结构。例如，

```
void f(Queue<X>& q)
try {
    for (;;) {
        X m = q.get(); // 如果队列空就抛出'Empty'
        // ...
    }
}
catch (Queue<X>::Empty) {
    return;
}
```

这样做确实有些诱惑力，对这种情况应该看成是错误或不看成错误，事情就不太清楚了。

与局部控制结构如`if`和`for`相比，异常处理是一种结构化更差的机制，通常在实际抛出异常时效率也更低。因此，应该只把异常机制用在那些使用常规控制结构很不优美甚至不可能的地方。注意，标准库在提供任意元素的`queue`时就没有使用异常（17.3.2节）。

对于结束检索函数的工作而言，采用异常作为另一种返回方式也可以是一种很优雅的技术——特别是在高度递归的检索函数里，譬如在一棵树中查找。看下面这个例子：

```
void fnd(Tree* p, const string& s)
{
    if (s == p->str) throw p; // 发现s
    if (p->left) fnd(p->left, s);
    if (p->right) fnd(p->right, s);
}

Tree* find(Tree* p, const string& s)
{
    try {
        fnd(p, s);
    }
    catch (Tree* q) { // q -> str == s
        return q;
    }
}
```

```

    }
    return 0;
}

```

然而，异常的这类应用很容易过度，并导致结构模糊的代码。在任何时候只要可能，我们还是应该坚持“异常处理就是错误处理”的观点。如果这样做了，在代码中就能清晰地划分为两类情况：正常代码和错误处理代码。这能使代码更容易理解。不幸的是，现实世界里并没有这样清晰的划分，程序的组织也将（而且应该在某种程度上）反应这一事实。

错误处理在本质上就是困难的。任何有助于维持一种关于一个错误是什么以及它被如何处理清晰的模型，都是非常宝贵的。

14.6 异常的描述

抛出或捕捉异常也对一个函数与其他函数的关系产生了影响。因此，将可能抛出的异常集合作为函数声明的一部分就有价值了。例如，

```
void f(int a) throw (x2, x3);
```

这说明`f()`只可能抛出两个异常`x2`和`x3`，以及从这些类型派生的异常，但不会抛出其他异常。当某个函数描述了它可能抛出的异常时，它也是有效地为其调用者提供了一种保证。如果在这个函数的执行中做了某种事情，试图废止所做出的保证，这个企图就会被转换为一个对`std::unexpected()`的调用。`unexpected()`的默认意义是`std::terminate()`，它通常将转而调用`abort()`。进一步的细节请参看9.4.1.1节。

在作用上

```

void f() throw (x2, x3)
{
    // 某些代码
}

```

等价于

```

void f()
try
{
    // 某些代码
}
catch (x2) { throw; }    // 重新抛出
catch (x3) { throw; }    // 重新抛出
catch (...) {
    std::unexpected();    // unexpected() 不会返回
}

```

这里最重要的优点在于函数的声明属于界面，而界面是函数的调用者可以看到的。而在另一方面，函数的定义则一般不是可用的东西。即使在我们能访问所有的库的源代码的情况下，我们也强烈地希望不要经常去查看它。此外，带有异常描述的函数也比手工写出的等价版本更短更清晰。

如果函数声明中不带异常描述，那么就假定它可能抛出任何异常。例如，

```
int f();    // 可能抛出任何异常
```

不抛出任何异常的函数可以用空表声明：

```
int g() throw ();    // 不会抛出异常
```

有人可能认为应该用默认方式表示函数不抛出异常。但是，这样规定实质上就是要求每个函数都写出异常描述，因此将导致大规模的重新编译，并将禁止与用其他语言写出的软件合作。这也会促使程序员设法去颠覆异常处理机制，写出欺骗性的代码去抑制异常。而这种情况又会给那些没有注意到这种颠覆的人们提供一种安全的假象。

14.6.1 对异常描述的检查

在编译时不可能捕捉到所有违反界面描述的情况。但是，编译时检查可以完成大部分工作。思考异常描述的方式应该是假定函数将要抛出它可能抛出的所有异常。编译时检查异常描述失效的规则很容易查出各种荒谬的情况。

如果一个函数的声明包含了异常描述，那么这个函数的每个声明（包括定义）都必须有一个包含着完全一致的异常类型集合的异常描述。例如，

```
int f() throw (std::bad_alloc);
int f()    // 错误：缺少异常描述
{
    // ...
}
```

有一点很重要：并没有要求跨过编译单位的边界对异常描述进行准确的检查。自然，某个实现可以做这种检查。然而，对许多大型和长期生存的系统而言有一件事也很重要：这类实现在做检查时，不要精心地给出一些刺耳的错误信息，而且只是在那些违背规则的情况不会在运行时被捕捉到的地方。

这里的要点是想保证，在某处增加一个异常时，不会强迫人们对相关的异常描述做完全的更新，也不会强制性地要求重新编译所有受到潜在影响的代码。一个系统应能在一种部分更新的状态中，依靠对未预期异常的动态（运行时）检查照常工作。这对于大规模系统的维护是必不可少的要求，因为在这里做大规模更新的代价极其昂贵，而且并不是所有的源代码都能使用。

要去覆盖一个虚函数，这个函数所带的异常描述必须至少是与那个虚函数的异常描述一样受限的（显式的或者隐式的）。看下面例子：

```
class B {
public:
    virtual void f();           // 可以抛出任何异常
    virtual void g() throw(X,Y);
    virtual void h() throw(X);
};

class D : public B {
public:
    void f() throw(X);          // ok
    void g() throw(X);          // 可以：D::g() 比B::g() 更受限
    void h() throw(X,Y);        // 错误：D::h() 不如B::h() 那么受限
};
```

这个规则其实不过是一种常识。如果派生类抛出了一个原函数并没有声言的异常，调用者当

然不可能准备去捕捉它。在另一方面，抛出的异常更少的覆盖函数显然会遵守由被覆盖函数所设定的异常描述。

与上述情况类似，你可以用一个指向具有更受限的异常描述的函数的指针，给一个指向带有不那么受限的异常描述的函数的指针赋值，但反过来就不行。例如，

```
void f() throw(X);
void (*pf1)() throw(X,Y) = &f;    // ok
void (*pf2)() throw() = &f;       // 错误: f() 不如pf2那么受限
```

特别地，你不能用一个指向没有异常描述的函数的指针，给一个指向带异常描述的函数的指针赋值：

```
void g(); // 可能抛出任何异常
void (*pf3)() throw(X) = &g;      // 错误: g() 不如pf3那么受限
```

异常描述并不是函数类型的一部分，*typedef*不能带有异常描述。例如，

```
typedef void (*PF)() throw(X);    // 错误
```

14.6.2 未预期的异常

异常描述可能导致调用*unexpected()*。通常不希望出现这种调用，除了在程序调试期间。通过细心地组织异常和界面描述可以避免这种调用。换一种方式，也可以拦截对*unexpected()*的调用，并使之无害化。

一个设计良好的子系统*Y*常常将它的所有异常都从一个类*Yerr*派生出来。例如，

```
class Some_Yerr : public Yerr { /* ... */};
```

有如下声明的函数

```
void f() throw(Xerr, Yerr, exception);
```

就能把所有的*Yerr*传给它的调用者。特别是*f()*可以通过将*Some_Yerr*传给其调用者的方式去处理它。这样，*f()*里的*Yerr*都不会触发*unexpected()*。

由标准库抛出的所有异常都是由*exception*派生出的（14.10节）。

14.6.3 异常的映射

偶尔也会有这种情况，在其中遇到一个未预期的异常就终止程序的策略显得过于严厉了。在这类情况下，就需要将*unexpected()*的行为修改为其他的能够接受的方式。

做到这一点的最简单方式就是将标准库异常*std::bad_exception*加入某个异常描述。在这种情况下，*unexpected()*将直接抛出一个*bad_exception*，而不是去调用某个试图应付困难的函数。例如，

```
class X{};
class Y{};

void f() throw(X, std::bad_exception)
{
    // ...
    throw Y();    // 抛出“坏”异常("bad" exception)
}
```

这个异常描述将捕捉未预期的异常Y，而后另外抛出一个***bad_exception***类型的异常。

bad_exception异常实际上也没有什么特别坏的地方；它只是提供了一种机制，没有调用***terminate()***那么严厉罢了。然而，它仍然是很粗鲁的，特别是丢失了引起问题的那个异常的所有信息。

14.6.3.1 异常的用户映射

现在来考虑一个为无网络环境写的函数***g()***，进一步假定***g()***被声明为带有一个异常描述，所以它只抛出与它的“子系统Y”有关的异常

```
void g() throw(Yerr);
```

现在假定我们要在一个网络环境里调用***g()***。

很自然，***g()***根本不可能知道有关网络的异常，当它遇到这类异常时，就会调用***unexpected()***。要将***g()***应用于分布式的环境，我们必须或者是提供代码去处理所有的网络异常，或者是重写***g()***。假定重写是不可行也是不希望做的事情，我们就可以通过重新定义***unexpected()***的意义的方式来处理这个问题。

存储耗尽的问题由***_new_handler***处理，它又由***set_new_handler()***确定。与此类似，对未预期异常的响应由***_unexpected_handler***决定，它又是通过***<exception>***中的***std::set_unexpected()***设置的：

```
typedef void(*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler);
```

要很好地处理未预期的异常，我们应首先采用“资源申请即初始化”技术，为***unexpected()***函数定义一个类：

```
class STC { // 保存和恢复类
    unexpected_handler old;
public:
    STC(unexpected_handler f) { old = set_unexpected(f); }
    ~STC() { set_unexpected(old); }
};
```

而后我们定义一个函数，使它具有我们所希望的***unexpected()***的意义。在目前情况中

```
class Yunexpected : public Yerr { };
void throwY() throw(Yunexpected) { throw Yunexpected(); }
```

在用做***unexpected()***函数时，***throwY()***将所有未预期的异常都映射为***Yunexpected***。

最后，我们提供***g()***的一个用于网络环境的版本

```
void networked_g() throw(Yerr)
{
    STC xx(&throwY); // 现在unexpected()抛出Yunexpected
    g();
}
```

因为***Yunexpected***是由***Yerr***派生出的，这里并没有违反异常描述。如果***throwY()***真的抛出一个违反异常描述的异常，那么就会调用***terminate()***了。

通过保存和恢复***_unexpected_handler***，我们就能使几个子系统都能控制对未预期异常的处理工作，而又不会相互影响。简而言之，这种将未预期的异常映射到预期异常的技术，也

就是系统以***bad_exception***形式提供的功能的一个更灵活版本。

14.6.3.2 找回异常的类型

把未预期的异常映射到***Yunexpected***，也就使***networked_g()***的用户能知道有一个未预期的异常被映射到了***Yunexpected***。但是，这个用户却无法知道被映射过来的到底是哪个异常。在***throwY()***中丢失了信息。采用一种简单技术，我们就能记录和传递有关的信息。继续前面的例子，我们可以用下面方式收集与***Network_exception***有关的信息：

```
class Yunexpected : public Yerr {
public:
    Network_exception* pe;
    Yunexpected(Network_exception* p) : pe(p?p->clone():0) {}
    ~Yunexpected() { delete pe; }
};

void throwY() throw(Yunexpected)
{
    try {
        throw; // 重新抛出将立即被捕捉!
    }
    catch (Network_exception& p) {
        throw Yunexpected(&p); // 抛出映射的异常
    }
    catch (...) {
        throw Yunexpected(0);
    }
}
```

重新抛出一个异常并再捕捉它，就使我们能够得到任何我们能命名的异常的一个句柄。函数***throwY()***由***unexpected()***调用，而***unexpected()***在概念上是由一个***catch(...)***处理器调用的。所以现在一定存在着某个能够重新抛出的异常。一个***unexpected()***函数不能忽略异常并退出。如果它真那样做的话，***unexpected()***本身就会抛出一个***bad_exception***（14.6.3节）。

函数***clone()***用于在自由存储中为异常分配一个新副本，在异常处理器清理局部变量后，这个副本还能继续存在。

14.7 未捕捉的异常

如果抛出的一个异常未被捕捉，那就会调用函数***std::terminate()***。当异常处理机制发现堆栈损坏，或者在由某个异常抛出而导致的堆栈回退过程中，被调用的析构函数企图通过抛出异常而退出时，也都会调用函数***std::terminate()***。

未预期的异常由***set_unexpected()***确定的***_unexpected_handler***处理。与此类似，对未捕捉的异常的响应由***_uncaught_handler***确定，而它由***<exception>***里的***std::set_terminate()***设置

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

返回值给出的是通过***set_terminate()***设置的前一个函数。

提出***terminate()***的原因是，在少数情况下，必须能用不那么精细的错误处理技术终止异常处理过程。例如，***terminate()***可能被用于结束某个进程，或甚至用于重新初始化一个系统。采用***terminate()***的意图是作为一种严格的限制，当由异常处理机制所实现的错误恢复策略失

败了，应该进入另一个容错策略层次时，就去应用它。

按照默认规定，`terminate()` 将调用 `abort()` (9.4.1.1节)。对大部分用户而言这都是一种正确选择，特别是在排除程序错误的阶段。

`_uncaught_handler` 被假定是不会返回其调用者的。如果它真的那样做，`terminate()` 就将调用 `abort()`。

注意，`abort()` 表示从程序中非正常退出。也可以用函数 `exit()` 终止程序，这个函数的返回值可以向外围系统表明程序是正常结束还是非正常结束 (9.4.1.1节)。

在程序因为未捕捉的异常而终止时，是否调用有关析构函数的问题将由具体实现决定。在一些系统上，不调用析构函数是至关重要的，这将使程序能够从排错系统中重新唤醒。在另一些系统里，其系统结构决定了在检索异常处理器的过程中，不调用那些析构函数是不可能的。

如果你希望在发生未捕捉异常时保证进行清理工作，你可以在所有真正需要关注的异常的处理器之外，再为 `main()` 添加一个捕捉一切的处理程序 (14.3.2节)。例如，

```
int main()
try {
    // ...
}
catch (std::range_error)
{
    cerr << "range error: Not again!\n";
}
catch (std::bad_alloc)
{
    cerr << "new ran out of memory\n";
}
catch (...) {
    // ...
}
```

这样就能捕捉到所有的异常，除了那些在全局变量的构造和析构中抛出的异常。没有办法一般性地捕捉全局变量初始化期间抛出的异常。对于非局部静态对象初始化中出现 `throw`，获得控制的惟一方法就是通过 `set_unexpected()` (14.6.2节)。这也是应该尽可能地避免全局变量的又一个原因。

当一个异常被捕捉到时，一般无法知道抛出它的确切位置。与排除系统对程序状态的了解相比，这种情况也表示了信息的丢失。在有些 C++ 开发环境里，对某些程序和某些人而言，这种情况也可能导致人们不去捕捉那些程序的设计本身无法恢复的异常。

14.8 异常和效率

原则上说，在不抛出异常的情况下，可以使异常处理的实现在运行时没有任何额外开销。另外，也可以做到使抛出异常并不总比调用函数的开销更大。在完成这些的同时，又能维持与 C 语言在调用序列、排错规范等方面的兼容性，又不引起显著的额外存储开销，这些都是可能的，但也非常困难。无论如何，应该记住那些能代替异常的东西也不是无代价的。在传统系统中，有一半代码实际上献身于错误处理也是很正常的情况。

考虑下面的简单函数 `f()`，看起来它与异常处理并无任何关系：

```

void g(int);
void f()
{
    string s;
    // ...
    g(1);
    g(2);
}

```

但是, `g()` 可能抛出异常, 所以 `f()` 必须包含一些代码, 使出现异常时 `s` 能够正确销毁。然而, 如果 `g()` 不采用抛出异常的方式, 它也得用另一种方式去报告出现的错误。因此, 可以与之比较的, 用常规方式写出的不用异常而又能够处理错误的代码, 就不是上面这样的简单代码, 而是类似下面这样的:

```

bool g(int);
bool f()
{
    string s;
    // ...
    if (g(1))
        if (g(2))
            return true;
        else
            return false;
    else
        return false;
}

```

人们通常不会这样系统化地处理错误, 当然, 事情也不总是重要到必须这样做的程度。但是, 在那些需要细心地系统化地处理错误的地方, 这样的簿记工作最好还是交给计算机去做, 也就是说, 利用异常处理机制。

异常描述 (14.6 节) 可能非常有助于改进所生成的代码。如果我们说明了 `g()` 不会抛出异常

```
void g(int) throw();
```

为 `f()` 生成的代码就可以改进。值得注意的是传统 C 函数都不会抛出异常, 所以, 在大部分程序里, 每个 C 函数都可以用空抛出描述 `throw()` 声明。特别地, 一个实现总知道只有很少的几个标准 C 库函数 (例如, `atexit()` 和 `qsort()`) 能够抛出异常, 它可以利用这个事实生成更好的代码。

在给一个 “C 函数” 空异常描述 `throw()` 之前, 要思考一下它是否有可能抛出异常。例如, 它可能已经转而去使用了 C++ 的 `new` 运算符 (该运算符可能抛出 `bad_alloc`), 或者它调用了可能抛出异常的 C++ 库。

14.9 处理错误的其他方式

异常处理机制的意图是提供一种方式, 使程序的一个部分可以通知另一部分, 它已经检查到一个 “异常的状况”。这里假定程序的两个部分是分别独立写出的, 而负责处理异常的那个程序部分常常可以对错误做出一些有意义的事情。

要想在程序里有效地使用异常处理器，我们需要一种整体策略，也就是说，程序的不同部分必须在如何使用异常、在哪里处理错误等诸方面保持一致。异常处理机制本质上就是非局部的，因此，与整体策略相符是最根本的要求。这也意味着，异常处理策略问题最好在设计的初始阶段予以考虑。它还意味着这个策略必须是简单（与这个程序的复杂性相比）而明晰的。在错误恢复这种本质上说就是很难处理的领域，根本就不可能始终如一地遵守一套很复杂的规则。

首先，认为某种单一机制或者技术能够处理所有错误的想法必须抛到一旁，这种想法必然导致复杂性。成功的容错系统只能是多层次的，每个层次在不至于导致过度扭曲的条件下，设法处理尽可能多的错误，将其余的错误留给更高的层次。*terminate()* 概念就是想支持这种观点，提供一种出口，供异常处理机制本身垮台，或者由于它的不完全应用而造成某个异常未被捕捉时使用。与此类似，*unexpected()* 概念也是为采用异常描述所提供的防火墙策略失败时提供的一个出口。

并不是每个函数都应该是一堵防火墙。在大部分系统中，将每个函数都写成能够完成充分的检查，保证它或者是成功结束，或者是以某种定义良好的方式失败，实际上根本不可行。这样做为什么不行的原因，对于不同的程序，不同的程序员都可能不同。然而，对于大型程序：

- [1] 为保证这种“可靠性”概念所需的工作量太大，以至根本不可能贯彻始终。
- [2] 时间空间上的额外开销太大，以至系统的运行情况无法接受（这里将存在一种反反复复检查同样错误的趋势，例如非法参数等）。
- [3] 用其他语言写出的功能未必能遵守这种规则。
- [4] 这种纯粹局部的“可靠性”概念带来的是“复杂性”，很可能实际变成整个系统的可靠性的障碍。

然而，将系统划分为一些独立的子系统，使它们或是成功结束，或是以某种定义良好的方式失败，则是至关重要的、可行的和经济的。这样，重要的库、子系统，或者关键性函数都应该按照这种方式设计。异常描述就是希望能成为这种库和子系统的界面。

我们常常无法享受从头开始设计整个系统的所有代码的乐趣。所以，在向系统的所有部分推行一种通行的错误处理策略时，我们也必须考虑那些采取了与我们不同的策略写出的程序片段。为了完成这种工作，我们必须去处理各种各样的利害关系：某个程序片的资源管理方式，它在发生一个错误后留给系统的状态等。这里的目标应该是让这种程序片看起来像是遵守通用的错误处理策略，即使其内部实际遵守的是另一套策略。

有时候，也可能需要将一种风格的错误报告转换为另一种报告。例如，我们可能要在调用一个C库函数之后检查*errno*，并可能抛出一个异常；或者反过来，在一个C++ 库里捕捉一个异常，并在返回C程序之前设置*errno*：

```
void callC() throw(C_blewit)
{
    errno = 0;
    c_function();
    if (errno) {
        // 清理，如果可能且必须
        throw C_blewit(errno);
    }
}
```

```
extern "C" void call_from_C() throw()
{
    try {
        c_plus_plus_function();
    }
    catch (...) {
        // 清理, 如果可能且必须
        errno = E_CPLPLFCTBLEWIT;
    }
}
```

在这些情况中, 最重要的就是足够系统化, 以保证错误报告风格的转换是完全的。

错误处理应该——尽可能地——层次化。如果某个函数检查出一个运行时错误, 它不应该要求其调用者为完成恢复或者资源申请提供任何帮助, 因为这种请求将造成系统中的循环依赖。这种情况的出现转而又使程序难于理解, 并有可能在错误处理和恢复代码中引进无穷循环。

使问题简单化的技术, 如“资源申请即初始化”, 使问题简单化的假设, 如“异常就表示错误”都应该尽可能地使用, 以使错误处理代码规范化。参看24.3.7.1节关于如何利用不变式和断言使异常的触发更加规范的有关想法。

14.10 标准异常

这里是标准异常, 以及抛出它们的函数、运算符和一般性功能:

标准异常 (由语言抛出)			
名字	抛出	参考	头文件
<i>bad_alloc</i>	<i>new</i>	6.2.6.2节、19.4.5节	<code><new></code>
<i>bad_cast</i>	<i>dynamic_cast</i>	15.4.1.1节	<code><typeinfo></code>
<i>bad_typeid</i>	<i>typeid</i>	15.4.4节	<code><typeinfo></code>
<i>bad_exception</i>	异常描述	14.6.3节	<code><exception></code>

标准异常 (由标准库抛出)			
名字	抛出	参考	头文件
<i>out_of_range</i>	<i>at()</i>	3.7.2节、16.3.3节、20.3.3节	<code><stdexcept></code>
	<i>bitset<>::operator[]()</i>	17.5.3节	<code><stdexcept></code>
<i>invalid_argument</i>	按位设置构造函数	17.5.3.1节	<code><stdexcept></code>
<i>overflow_error</i>	<i>bitset<>::to_ulong()</i>	17.5.3.3节	<code><stdexcept></code>
<i>ios_base::failure</i>	<i>ios_base::clear()</i>	21.3.6节	<code><ios></code>

所有库异常都是同一个类层次结构中的部分, 这个结构的根是标准库异常类*exception*, 在头文件`<exception>`里给出

```
class exception {
public:
    exception() throw();
```

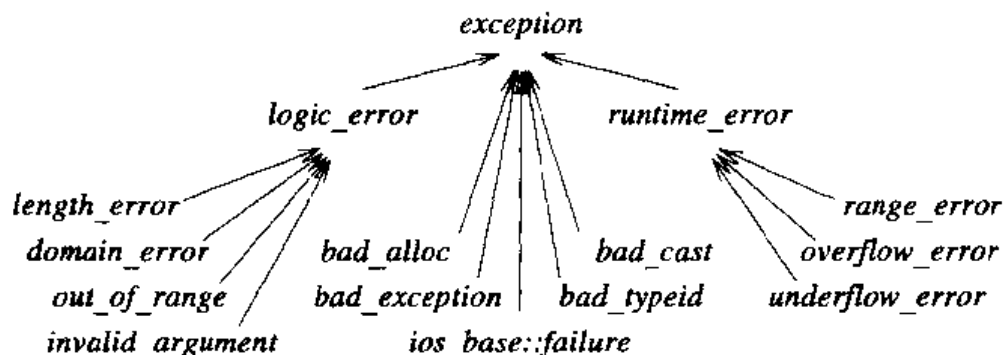
```

exception(const exception&) throw();
exception& operator=(const exception&) throw();
virtual ~exception() throw();

virtual const char* what() const throw();
private:
    // ...
};

```

该类层次结构看起来的样子是



采取这种方式组织起这八个标准异常确实是颇费苦心的。这个层次结构是想为标准库之外定义的异常提供一个框架。逻辑错误是那些原则上能够在程序执行之前，或者通过检测函数和构造函数的参数捕捉到的错误。运行时错误则包括所有其他错误。有些人将此看做是对所有错误和异常都有用的框架；我并不这么看。

这些标准库异常类都没有向**exception**提供的函数集里添加任何函数，它们只是适当地定义了所需的虚函数。这样，我们可以写

```

void f()
try {
    // 使用标准库
}
catch (exception& e) {
    cout << "standard library exception " << e.what() << '\n';    // 好，或许
    // ...
}
catch (...) {
    cout << "other exception\n";
    // ...
}

```

所有标准异常都由**exception**派生。然而，并不是所有的异常都如此，所以，如果想通过捕捉**exception**去试图捕捉所有异常就完全是个错误想法。类似地，假定每个从**exception**派生出的异常都是标准库异常也是一种错误看法：程序员同样可以把他们自己的异常加入到这个层次结构之中。

注意，**exception**的那些操作本身并不抛出异常。特别地，这也意味着抛出一个标准库异常不会导致**bad_alloc**异常。异常处理机制为自己维持着一点为保存异常所用的内存（可能在堆栈上）。自然，完全可能写出代码，使之最终能消耗掉系统里的所有内存，这将导致程序失败。举个例子，下面是一个函数，如果调用，它将查出到底是函数调用还是异常处理机制先

用光了可用的存储:

```
void perverted()
{
    try {
        throw exception(); // 递归异常抛出
    }
    catch (exception& e) {
        perverted(); // 递归函数调用
        cout << e.what();
    }
}
```

这里的输出语句的用途就是为了防止编译器去重复使用由异常e所占据的内存。

14.11 忠告

- [1] 用异常做错误处理; 14.1节、14.5节、14.9节。
- [2] 当更局部的控制机构足以应付时, 不要使用异常; 14.1节。
- [3] 采用“资源申请即初始化”技术去管理资源; 14.4节。
- [4] 并不是每个程序都要求具有异常时的安全性; 14.4.3节。
- [5] 采用“资源申请即初始化”技术和异常处理器去维持不变式; 14.3.2节。
- [6] 尽量少用try块, 用“资源申请即初始化”技术, 而不是显式的处理器代码; 14.4节。
- [7] 并不是每个函数都需要处理每个可能的错误; 14.9节。
- [8] 在构造函数里通过抛出异常指明出现失败; 14.4.6节。
- [9] 在从赋值中抛出异常之前, 使操作对象处于合法状态; 14.4.6.2节。
- [10] 避免从析构函数里抛出异常; 14.4.7节。
- [11] 让main()捕捉并报告所有的异常; 14.7节。
- [12] 使正常处理代码和错误处理代码相互分离; 14.4.5节、14.5节。
- [13] 在构造函数里抛出异常之前, 应保证释放在此构造函数里申请的所有资源; 14.4节。
- [14] 使资源管理具有层次性; 14.9节。
- [15] 对于主要界面使用异常描述; 14.9节。
- [16] 当心通过new分配的内存存在发生异常时没有释放, 并由此而导致存储的流失; 14.4.1节、14.4.2节、14.4.4节。
- [17] 如果一函数可能抛出某个异常, 就应假定它一定会抛出这个异常; 14.6节。
- [18] 不要假定所有异常都是由exception类派生出来的; 14.10节。
- [19] 库不应该单方面终止程序。相反, 应该抛出异常, 让调用者去做决定; 14.1节。
- [20] 库不应该生成面向最终用户的错误信息。相反, 它应该抛出异常, 让调用者去做决定; 14.1节。
- [21] 在设计的前期开发出一种错误处理策略; 14.9节。

14.12 练习

1. (*2) 将STC类(14.6.3.1节)推广为一个模板, 让它采用“资源申请即初始化”技术为各种类型保存和重置函数。

2. (*3) 将取自11.11节的*Ptr_to_T*类做成模板，让它通过异常发出运行时错误信号。
3. (*3) 写一个函数*find*，它在一个基于*char** 域的结点的二叉树中通过匹配做检索。如果找到了一个包含*hello*的结点，*find("hello")* 就返回一个指向该结点的指针。用异常表明“没找到”。
4. (*3) 定义类*Int*，它的行为与内部类型*int*一模一样，只是在出现溢出时抛出异常。
5. (*2.5) 从C与你所用操作系统的界面取来打开、关闭、读、写文件的函数，给出与之等价的C++函数，其中调用对应的C函数，但在出现错误时抛出异常。
6. (*2.5) 写一个带有*Range*和*Size*异常的完整的*Vector*模板。
7. (*1) 写一个循环，计算14.12[6] 中定义的*Vector*之和，然而却不检查*Vector*的大小。为什么这是一个很坏的主意？
8. (*2.5) 考虑用类*exception*作为所有表示异常的类的基类。结果看起来会是什么样子？它应该如何使用？这样做有什么好处？要求使用这样的类可能带来什么不利情况？
9. (*1) 给了：

```
int main() { /* ... */ }
```

修改它，使它能捕捉所有异常并将它们变成错误信息，而后*abort()*。提示：14.9节里的*call_from_C()* 不能完全处理所有的情况。

10. (*2) 写一个适用于实现回调的类或者模板。
11. (*2.5) 为某个系统所支持的并发写一个*Lock*类。

第15章 类层次结构

抽象就是有选择地装糊涂。

——Andrew Koenig

多重继承——歧义性解析——继承和使用声明——重复的基类——虚基类——多重继承的使用——访问控制——保护——访问基类——运行时类型信息——*dynamic_cast*——静态和动态强制——从虚基类强制——*typeid*——扩充的类型信息——运行时类型信息的使用和误用——指向成员的指针——自由存储——虚构造函数——忠告——练习

15.1 引言和概述

本章将讨论派生类和虚函数如何与其他语言功能相互作用，例如访问控制、名字查找、自由存储管理、构造函数、指针和类型转换等。它包括五个主要部分：

15.2节 多重继承

15.3节 访问控制

15.4节 运行时类型识别

15.5节 指向成员的指针

15.6节 自由空间的使用

总而言之，一个类是从一些基类的格^①构造出来的。因为大多数这种格实际上是树，一个类格也常常被称做类层次结构。我们应设法设计这些类，使得用户不必过度操心某个类是如何由另外的类组合而成的。特别是虚函数调用机制，它能保证当我们对某个对象调用函数 $f()$ 时，无论在层次结构中的类有多少函数提供了用于调用 $f()$ 的声明，为这个对象调用的总是同一个函数。本章将集中讨论组织类层次结构和对这些类各个部分的访问进行控制的方法，以及编译时和运行时在类层次结构中漫游的功能。

15.2 多重继承

正如2.5.4节和12.3节所示，一个类可以有多于一个直接基类，也就是说，可以在类声明的：之后给定多个类名。考虑一个模拟，其中并行的活动由类***Task***表示，数据收集与显示则由类***Displayed***完成。此后我们就可以定义用于模拟有关实体的类，如类***Satellite***

```
class Satellite : public Task, public Displayed {  
    // ...  
};
```

① 格 (lattice)：图论中的术语。实际上，C++中类的继承结构未必会形成一个格，一般而言是一个有向无环图 (directed acyclic graph)。下面还有几处提到格，也都应该改为有向无环图。——译者注

采用多个直接基类的情况通常称为多重继承。与此相对应，只有一个直接基类称为单继承。

除了那些特别为*Satellite*描述的操作之外，在*Task*和*Displayed*中所有操作的并集都可以使用，例如，

```
void f(Satellite& s)
{
    s.draw();      // Displayed::draw()
    s.delay(10);   // Task::delay()
    s.transmit();   // Satellite::transmit()
}
```

类似的，可以将一个*Satellite*传递给那些期望*Task*或*Displayed*的函数。例如，

```
void highlight(Displayed*);
void suspend(Task*);

void g(Satellite* p)
{
    highlight(p);   // 传递一个指向Satellite的Displayed部分的指针
    suspend(p);     // 传递一个指向Satellite的Task部分的指针
}
```

这些东西的实现明显涉及到一些（简单的）编译技术，以便保证期望*Task*的函数看到的该*Satellite*的那个部分与期望*Displayed*的函数所看到的不同。虚函数的工作如常。例如，

```
class Task {
    // ...
    virtual void pending() = 0;
};

class Displayed {
    // ...
    virtual void draw() = 0;
};

class Satellite : public Task, public Displayed {
    // ...
    void pending();      // 覆盖 Task::pending()
    void draw();         // 覆盖 Displayed::draw()
};
```

这样就能保证，对于被当做*Task*和*Displayed*的*Satellite*，实际使用的仍将是*Satellite::draw()*和*Satellite::pending()*。

请注意，如果（只）用单继承，程序员在实现类*Displayed*、*Task*和*Satellite*时的可能选择就会受到限制。一个*Satellite*可以是一个*Task*或者一个*Displayed*，但不能同时为两者（除非*Task*是由*Displayed*派生，或者相反）。任何一种选择都将涉及到灵活性方面的损失。

为什么有人会希望有*Satellite*类？与有些人的怀疑相反，*Satellite*类的例子是真实的。确实存在过——或许现在还存在着——这样一个程序，它就是按照上面描述的方式利用多重继承构造起来的。这个程序被用于研究涉及人造卫星、地面站等的通信系统的设计。有了这种模拟，我们就能回答有关通信流量的问题，确定在某个地面站被雷雨阻塞时的正确响应方式，考虑卫星连接和地面连接之间的平衡折中，如此等等。这种模拟肯定要涉及到各种显示和排除错误的操作。同时，我们也必须去存储诸如*Satellite*一类的对象以及它们的子部分的状态，以便进行分析、排除程序错误、做出错时的恢复。

15.2.1 歧义性解析

两个基类中可能出现同样名字的成员函数。比如说

```
class Task {
    // ...
    virtual debug_info* get_debug();
};

class Displayed {
    // ...
    virtual debug_info* get_debug();
};
```

在用到一个*Satellite*时,就必须消解有关这些函数的歧义性问题

```
void f(Satellite* sp)
{
    debug_info* dip = sp->get_debug(); // 错误: 歧义
    dip = sp->Task::get_debug();       // ok
    dip = sp->Displayed::get_debug();   // ok
}
```

然而,通过明确写出的方式消除歧义显得比较混乱,最好是通过在派生类里定义新函数的方式消解这类问题

```
class Satellite : public Task, public Displayed {
    // ...
    debug_info* get_debug() // 覆盖Task::get_debug() 和 Displayed::get_debug()
    {
        debug_info* dip1 = Task::get_debug();
        debug_info* dip2 = Displayed::get_debug();
        return dip1->merge(dip2);
    }
};
```

这就使有关*Satellite*的基类的信息局部化了。由于*Satellite::get_debug()*覆盖了来自其两个基类的*get_debug()*函数,无论何时,只要对*Satellite*对象调用*get_debug()*,实际调用的就是*Satellite::get_debug()*。

加上限定词的名字*Telstar::draw*将能引用在*Telstar*内部的或者它的某个基类中声明的*draw*。例如,

```
class Telstar : public Satellite {
    // ...
    void draw()
    {
        draw(); // 呜呼! 递归调用
        Satellite::draw(); // 找到Displayed::draw
        Displayed::draw();
        Satellite::Displayed::draw(); // 多余的重复限定
    }
};
```

也就是说,如果某个*Satellite::draw()*不能解析为在*Satellite*里声明的某个*draw*,那么编译器

就会到它的基类中去找，也就是说，去寻找`Task::draw`和`Displayed::draw`。如果正好找到了一个匹配，就使用它。否则，`Satellite::draw`或者未找到，或者是有歧义。

15.2.2 继承和使用声明

重载解析的使用不会跨越不同类的作用域（7.4节）。特别地，来自不同基类的函数之间的歧义性不能基于参数类型完成解析。

组合起一些基本无关的类，例如`Stellite`示例中的`Task`和`Displayed`，这时名字的类似性未必表明了同样的用途。在出现了这种名字冲突时，它们通常总会在程序员那里引起一点惊奇。例如，

```
class Task {
    // ...
    void debug(double p);    // 只对比p优先级低的情况打印信息
};

class Displayed {
    // ...
    void debug(int v);    // v的优先级越高，打印的信息越多
};

class Satellite : public Task, public Displayed {
    // ...
};

void g(Satellite* p)
{
    p->debug(1);           // 错误：歧义。是Displayed::debug(int) 还是Task::debug(double)?
    p->Task::debug(1);      // ok
    p->Displayed::debug(1); // ok
}
```

如果在不同基类中使用同样名字是一种精心筹划的设计决策，或者用户希望能够基于实际参数类型做出选择，那么又该怎么办呢？在这些情况下，可以用使用声明（8.2.2节）将这些函数引进一个公共的作用域中。例如，

```
class A {
public:
    int f(int);
    char f(char);
    // ...
};

class B {
public:
    double f(double);
    // ...
};

class AB : public A, public B {
public:
    using A::f;
    using B::f;
    char f(char);    // hides A::f(char)
    AB f(AB);
};
```

```

void g (AB& ab)
{
    ab.f(1);           // A::f(int)
    ab.f('a');         // AB::f(char)
    ab.f(2.0);         // B::f(double)
    ab.f(ab);          // AB::f(AB)
}

```

使用声明使程序员可以组合起一组来自不同基类的和来自派生类的重载函数。在派生类里声明的函数就会遮蔽在基类里那些原本可用的函数，来自基类的虚函数还是像以往一样可以覆盖（15.2.3.1节）。

在一个类定义里的使用声明（8.2.2节）所引用的必须是基类的成员。不能通过使用声明取用这个类、派生它的类以及它们的成员之外的某个类的成员。使用指令（8.2.3节）不能出现在类定义里，也不能对一个类使用。

使用声明也不能用于获取对于更多的信息的访问权，它只是一种使可访问信息能够以更方便的方式使用的机制（15.3.2.2节）。

15.2.3 重复的基类

有了能描述多个基类的能力，也就出现了将一个类两次作为基类的可能性。举例来说，如果**Task**和**Displayed**都是由**Link**派生而来，在一个**Satellite**里就会出现两个**Link**

```

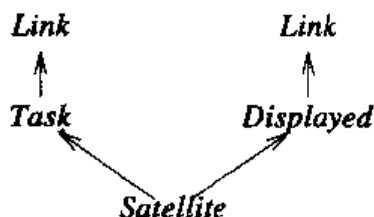
struct Link {
    Link* next;
};

class Task : public Link {
    // 这个Link用于维护Task表（调度表）
    // ...
};

class Displayed : public Link {
    // 这个Link用于维护所有Displayed对象的表（显示表）
    // ...
};

```

这并不会引起任何问题。两个互相独立的**Link**用于表示不同的链接，而且两个表也不会相互干扰。自然，如果要引用**Link**类的成员，就有出现歧义性的危险（15.2.3.1节）。画出的一个**Satellite**对象具有下面的样子：



对有些示例而言，其公共基类不应该表示为两个分离对象，这种情况可以用虚基类来处理（15.2.4节）。

通常，像**Link**这样一个基类重复出现的情况只是一个实现细节，不应该在其直接派生类之外使用。如果这样的一个基类必须在某个地方引用，而在那里能够看到多于一个基类的副

本时，有关引用就必须显式加以限定，以消解歧义性。例如，

```
void mess_with_links(Satellite* p)
{
    p->next = 0;           // 错误：歧义性（哪个Link？）
    p->Link::next = 0;      // 错误：歧义性（哪个Link？）
    p->Task::next = 0;      // ok
    p->Displayed::next = 0; // ok
    // ...
}
```

这也正是在消解对成员的歧义性引用时所采用的机制（15.2.1节）。

15.2.3.1 覆盖

在重复基类中的虚函数可以由派生类里的（一个）函数覆盖。例如，有人可能按如下方式表示一个对象具有从文件读入自己，以及将自己写入文件的能力：

```
class Storable {
public:
    virtual const char* get_file() = 0;
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable() {}
};
```

很自然，若干程序员可能会基于这个类开发出另一些类，使它们可以独立地或者组合式地构造出各种更精致的类。例如，停止或重新启动模拟的一种方式就是将模拟中的成分存储起来，而后在需要时再恢复之。这种想法可能如下实现

```
class Transmitter : public Storable {
public:
    void write();
    // ...
};

class Receiver : public Storable {
public:
    void write();
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    const char* get_file();
    void read();
    void write();
    // ...
};
```

在典型情况下，覆盖函数将调用基类中的版本，而后再做一些有关派生类的特殊工作：

```
void Radio::write()
{
    Transmitter::write();
    Receiver::write();
    // 写Radio的特殊信息
}
```

从重复的基类强制到派生类的问题将在15.4.2节讨论。关于一种用来自不同派生类的不同函数覆盖各个*write()*的技术, 请见25.6节。

15.2.4 虚基类

上一节的*Radio*例子可以工作, 因为类*Storable*可以安全、方便和有效地重复出现。对于构造作为其他类的良好构件的各种类而言, 这种情况并不常见。举个例子, 我们可能定义*Storable*来保存文件名, 以便在这些文件里存储对象:

```
class Storable {
public:
    Storable(const char* s);
    virtual void read() = 0;
    virtual void write() = 0;
    virtual ~Storable();
private:
    const char* store;

    Storable(const Storable&);
    Storable& operator=(const Storable&);
};
```

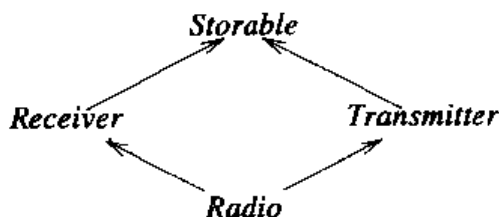
要使用这个看来只是稍加修改了的*Storable*, 我们就必须修改*Radio*的设计。一个对象的所有成分必须共享*Storable*的同一个副本, 否则, 要避免同一个对象被存储了多个副本的情况就会变得非常困难。描述这种共享的一种机制是虚基类。在任何派生类中的*virtual*基类总用同一个(共享)对象表示。例如,

```
class Transmitter : public virtual Storable {
public:
    void write();
    // ...
};

class Receiver : public virtual Storable {
public:
    void write();
    // ...
};

class Radio : public Transmitter, public Receiver {
public:
    void write();
    // ...
};
```

图形表示是



将这个图与15.2.3节中*Satellite*对象的图做一个比较, 就可以看到常规继承和虚继承之间的差

异了。在一个继承图里，每个名字用**virtual**刻画的基类，将总是用这个类的同一个对象表示。在另一方面，没有用**virtual**描述的每个基类都有定义自己的子对象去表示它。

15.2.4.1 用虚基类的程序设计

在为存在虚基类的类定义函数时，一般说程序员并不知道这个基类是与其他派生类共享的。在实现某种服务，其中要求调用基类的某个函数恰好一次时，这种情况就可能引起问题。例如，语言保证对于一个虚基类的构造函数将调用恰好一次。虚基类的构造函数将（隐式地或者显式地）从完整对象的构造函数（最终派生类的构造函数）里调用。例如，

```
class A { // 无构造函数
    // ...
};

class B {
public:
    B(); // 默认构造函数
    // ...
};

class C {
public:
    C(int); // 无默认构造函数
};

class D : virtual public A, virtual public B, virtual public C
{
    D() { /* ... */ } // 错误：没有对C的默认构造函数
    D(int i) : C(i) { /* ... */ }; // ok
    // ...
};
```

虚基类的构造函数将在其派生类的构造函数之前调用。

在需要之处，程序员可以模拟这种模式，方法就是只从最终派生类里调用虚基类的函数。考虑一个例子，假定我们有一个基本的**Window**（窗口）类，它知道如何画出自己的内容：

```
class Window {
    // 基本功能
    virtual void draw();
};
```

此外，我们还有多种方式去修饰窗口，而且可以为它增加功能：

```
class Window_with_border : public virtual Window {
    // 边框功能
    void own_draw(); // 显示边框
    void draw();
};

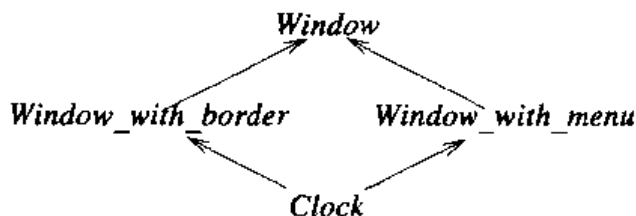
class Window_with_menu : public virtual Window {
    // 菜单功能
    void own_draw(); // 显示菜单
    void draw();
};
```

函数**own_draw()**不必是虚函数，因为它们的目的就是在一个虚函数**draw()**里使用，该函数“知道”它的调用所针对的那个对象的类型。

从这里出发，我们可以组织起一个可行的**Clock**（时钟）类

```
class Clock : public Window_with_border, public Window_with_menu {
    // 时钟功能
    void own_draw(); // 显示钟面和指针
    void draw();
};
```

用图形表示是



现在就可以借助`own_draw()`函数写出`draw()`函数了, 可以使任何对`draw()`的调用者都只能让`Window::draw()`被调用一次。做到这些并不依赖于到底对哪种`Window`调用`draw()`

```
void Window_with_border::draw()
{
    Window::draw();
    own_draw(); // 显示边框
}

void Window_with_menu::draw()
{
    Window::draw();
    own_draw(); // 显示菜单
}

void Clock::draw()
{
    Window::draw();
    Window_with_border::own_draw();
    Window_with_menu::own_draw();
    own_draw(); // 显示钟面和指针
}
```

从虚基类向派生类强制的问题将在15.4.2节讨论。

15.2.5 使用多重继承

多重继承的最简单最明显的应用, 就是利用它将两个原本不相干的类“粘合”起来, 作为第三个类的实现的一部分。在15.2节里基于`Task`类和`Displayed`类构造出`Satellite`类就是这方面的一个例子。多重继承的这种应用是生硬的、有效的, 但也是不那么令人感兴趣的。简而言之, 它就是使程序员省去了写许多前推函数的麻烦。这种技术不会明显地影响一个程序的总体设计, 偶然的还会与保持实现细节隐蔽的希望相冲突。然而, 一种技术也不一定要非常巧妙才能有用处。

利用多重继承为抽象类提供实现则是更根本性的情况, 它将会影响程序的设计方式。12.4.3节的`BB_ival_slider`类是一个例子

```
class BB_ival_slider
: public Ival_slider // 界面
, protected BBslider // 实现
```

```

{
    // Ival_slider和BBslider所需要的函数的实现
    // 使用BBslider提供的功能
};

```

在这个例子里，两个基类扮演着逻辑上不同的角色。一个基类作为公用的抽象基类，提供了一个界面；另一个则是受保护的具类，提供了实现的“细节”。这两种角色反应了类的不同风格和所提供的访问控制。多重继承在这里的使用几乎是本质性的，因为派生类需要去覆盖来自界面和来自实现的函数。

多重继承也使兄弟类之间能够共享信息，而又不会在程序里引进对同一基类的依赖性。这也就是那种通常被称做钻石形继承的情况（例如，15.2.4节的**Radio**和15.2.4.1节**Clock**）。如果基类不应该重复，那么就需要采用虚基类，而不是常规基类。

我发现，如果虚基类或者由虚基类直接派生的类是抽象类，钻石形继承将特别容易控制。比如说，再考虑一下12.4节**Ival_box**类的例子。在最后，我将所有的**Ival_box**都做成了抽象的，以反映它们作为纯粹界面的角色。这样做就使我能将所有的细节都放到特殊的实现类里去。另外，在为实现所用的经典窗口系统的类层次结构中，也做到了所有实现细节的共享。

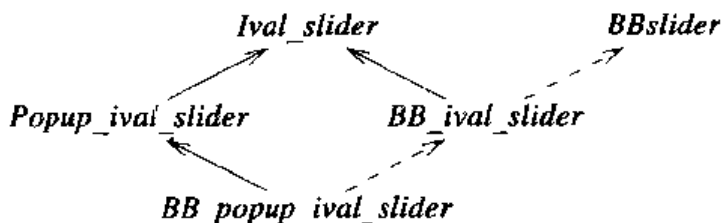
让实现**Popup_ival_slider**的类能够共享实现普通**Ival_slider**的类中的大部分实现，这也是一件有意义的事情，归根结底，这些类可以共享除了用户提示处理之外的所有东西。然而，避免**Ival_slider**对象在作为结果的滑块实现对象里重复出现是一个很显然的问题。为此，我们就应该把**Ival_slider**作为虚基类

```

class BB_ival_slider : public virtual Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public virtual Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public virtual Popup_ival_slider, protected BB_ival_slider { /* ... */ };

```

图形表示是



很容易设想由**Popup_ival_slider**进一步派生出的界面，以及由这些类和**BB_popup_ival_slider**进一步派生出的实现类。

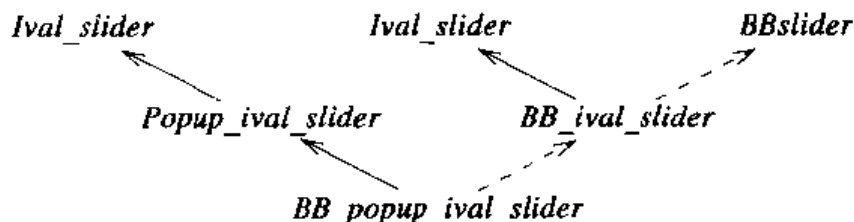
如果我们将这种思想进一步推到它的逻辑结论，那就是，组成我们应用的界面的抽象类的所有派生都应该是虚的。这确实是一个看起来最符合逻辑，最具有--般性，也最灵活的解决问题方案。我没有这样做的原因部分是由于历史，部分是因为实现虚基类的最明显、最通用技术所带来时间和空间上的额外开销，这也是将这种技术广泛用于类设计并不那么吸引人的原因。对于一个原本很有吸引力的设计，这种开销应该成为问题吗？可以注意到，表示**Ival_slider**的对象通常仅保存着一个虚表指针。正如在15.2.4节指出的，这种抽象类不保存任何数据，重复了也不会有任何影响。这样，我们就可以去掉虚基类，改用常规基类

```

class BB_ival_slider : public Ival_slider, protected BBslider { /* ... */ };
class Popup_ival_slider : public Ival_slider { /* ... */ };
class BB_popup_ival_slider
    : public Popup_ival_slider, protected BB_ival_slider { /* ... */ };

```

图形表示是



这特别像承认了前面所给出的那个更清晰方式之后的一个可行性优化。但也出现了一个潜在问题，现在一个 *BB_popup_ival_slider* 就不能隐式地转换到一个 *Ival_slider* 了。

15.2.5.1 覆盖虚基类的函数

派生类可以覆盖其直接或者间接基类的虚函数。特别地，两个不同的类也可能覆盖了来自虚基类的不同虚函数。按照这种方式，就可以用几个派生类为实现一个虚基类给出的界面服务。举个例子。*Window* 类可能有函数 *set_color()* 和 *prompt()*。在这种情况下，类 *Window_with_border* 可能覆盖 *set_color()* 函数，作为它自己对颜色模式进行控制的一部分；类 *Window_with_menu* 可能覆盖 *prompt()*，作为其控制用户交互动作的一部分：

```

class Window {
    // ...
    virtual void set_color(Color) = 0;    // 设置背景颜色
    virtual void prompt() = 0;
};

class Window_with_border : public virtual Window {
    // ...
    void set_color(Color);    // 控制背景颜色
};

class Window_with_menu : public virtual Window {
    // ...
    void prompt(); // 控制用户交互动作
};

class My_window : public Window_with_menu, public Window_with_border {
    // ...
};

```

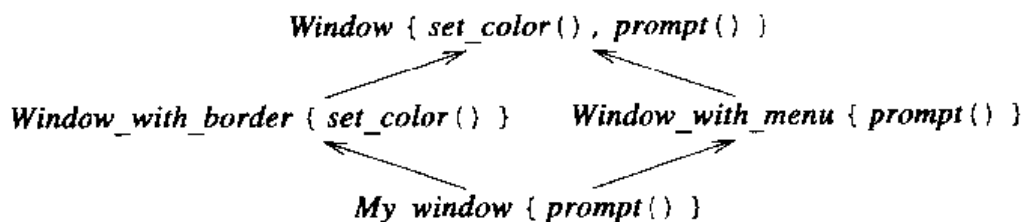
如果不同的派生类覆盖了同一个函数，那又会怎么样呢？当且仅当某个覆盖类是从覆盖此函数的每个类派生出时，才允许这样做。也就是说，必须有一个函数覆盖了所有其他的覆盖函数。例如，*My_window* 可能覆盖 *prompt()*，以改善 *Window_with_menu* 提供的功能

```

class My_window : public Window_with_menu, public Window_with_border {
    // ...
    void prompt(); // 不将用户交互事宜留给基类
};

```

对应图形是



如果两个类覆盖了同一个基类函数，但它们又互不覆盖，这个类层次结构就是错的。对此将无法构造出虚函数表，因为在最终完成的对象上，对这个函数的调用将是有歧义的。例如，假设15.2.4节的`Radio`里没有声明`write()`，在定义`Radio`时，`Receiver`和`Transmitter`里的`write()`声明将导致一个错误。就像在`Radio`里所做的那样，可以通过在最终派生类里加入一个覆盖函数的方式来消除这种冲突。

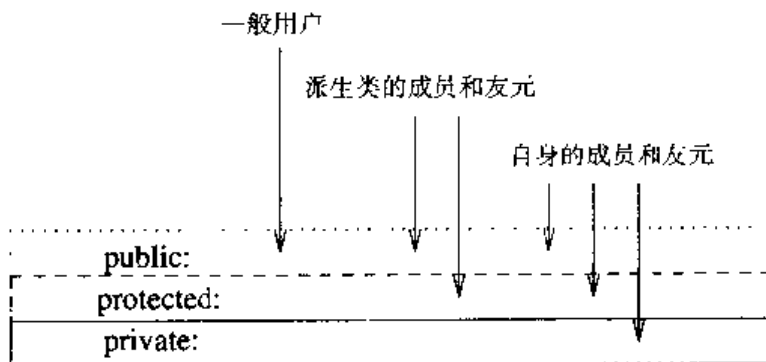
一个为虚基类提供部分实现——但不提供全部实现——的类通常被称为“混人类”。

15.3 访问控制

类中的一个成员可以是`private`、`protected`或`public`：

- 如果一个成员是`private`，它的名字将只能由其声明所在类的成员函数和友元使用。
- 如果一个成员是`protected`，它的名字只能由其声明所在类的成员函数和友元，以及由该类的派生类的成员函数和友元使用（见11.5节）。
- 如果一个成员是`public`，它的名字可以由任何函数使用。

这些反应了一种观点，存在着访问一个类的三类函数：实现这个类的函数（该类的成员和友元），实现派生类的函数（派生类的成员和友元），还有其他函数。这种情况可以用如下的图形表示：



这种访问控制被一致性地用于所有的名字，一个名字引用的是什么并不影响对它的使用控制。这也就意味着，我们除可以有私用数据成员之外，还可以有私用成员函数、类型、常量等。例如，有效的非侵入式（16.2.1节）表类通常都需要用一个数据结构保存元素的踪迹。这种信息最好是作为私用的。

```

template<class T> class List {
private:
    struct Link { T val; Link* next; };
    struct Chunk {
        enum { chunk_size = 15 };
        Link v[chunk_size];
        Chunk* next;
    };
};
  
```

```

};
Chunk* allocated;
Link* free;
Link* get_free();
Link* head;
public:
    class Underflow { }; // 异常类

    void insert(T);
    T get();
    // ...
};

template<class T> void List<T>::insert(T val)
{
    Link* lnk = get_free();
    lnk->val = val;
    lnk->next = head;
    head = lnk;
}

template<class T> List<T>::Link* List<T>::get_free()
{
    if (free == 0) {
        // 分配一个新块，将其中所有Link放入自由表
    }
    Link* p = free;
    free = free->next;
    return p;
}

template<class T> T List<T>::get()
{
    if (head == 0) throw Underflow();

    Link* p = head;
    head = p->next;
    p->next = free;
    free = p;
    return p->val;
}

```

在成员函数定义时写`List<T>::`就进入了`List<T>`的作用域。因为在提到`List<T>::get_free()`之前就提到了`get_free()`的返回类型，所以必须使用完全的名字`List<T>::Link`，而不能只用简单的`Link`。

非成员函数（除了友元之外）没有这种访问权：

```

void would_be_meddler(List<T>* p)
{
    List<T>::Link* q = 0;           // 错误: List<T>::Link 是私用的
    q = p->free;                     // 错误: List<T>::free 是私用的
    // ...
    if (List<T>::Chunk::chunk_size > 31) { // 错误: List<T>::Chunk::chunk_size 是私用的
        // ...
    }
}

```

在`class`里，成员默认为`private`，而`struct`里的成员默认为`public`（10.2.8节）。

15.3.1 保护成员

考虑15.2.4.1节的`Window`示例，其中函数`own_draw()`的设计就是为了作为构造块供派生类使用，对一般性的使用而言是不安全的。在另一方面，`draw()`操作的设计则是为了一般性使用。这一区分通过将`Window`类的界面划分为两个部分的方式表述：*protected*界面和*public*界面

```
class Window_with_border {
public:
    virtual void draw();
    // ...
protected:
    void own_draw();
    // 其他为创建工具而用的功能
private:
    // 表示等
};
```

派生类只能为它的类型的对象访问基类的保护成员

```
class Buffer {
protected:
    char a[128];
    // ...
};

class Linked_buffer : public Buffer { /* ... */ };

class Cyclic_buffer : public Buffer {
    // ...
    void f(Linked_buffer* p) {
        a[0] = 0;        // 可以：访问cyclic_buffer自己的保护成员
        p->a[0] = 0;     // 错误：访问不同类型的保护成员
    }
};
```

这就能防止由于一个派生类破坏了属于另一个派生类的数据而产生的一些微妙错误。

15.3.1.1 保护成员的使用

数据隐藏的简单公用/私用模型能很好地满足具体类型概念的需要（10.3节）。然而，在有了派生类之后，一个类就有了两种用户：派生类和“一般公众”。实现类的操作的成员和友元代表着这些用户在这个类的对象上的操作。公用/私用模型使程序员能够很清晰地划分实现者和一般公众，但却没有提供一种方法，以迎合派生类的特殊需要。

声明为*protected*的成员的滥用远比声明为*private*的成员多得多。应特别指出，声明一些保护数据成员通常都是设计错误。将位于一个公共类中的大量数据提供给派生类使用，将使这些数据对破坏敞开了大门。更糟的是，保护数据与公用数据一样很难重新构建，因为没有办法去找到对它们的所有使用。这就会使保护数据变成软件维护中的问题。

幸好，你并不必须采用保护数据；*private*是类中的默认情况，通常也是更好的选择。在我的经验里，对于将大量数据放在基类里供派生类直接使用的情况，总存在着另外一些的替代方式。

请注意，所有这些批评对*protected*成员函数都不重要，*protected*是描述供派生类使用的操作的极好方式。12.4.2节的`Ival_slider`是这方面的一个示例。如果这个例子中的实现类是

private，进一步的派生就无法进行了。

阐释访问各种成员的技术性示例可以在C.11.1节找到。

15.3.2 对基类的访问

像成员一样，基类也可以是*private*、*protected*或*public*：

```
class X : public B { /* ... */ };
class Y : protected B { /* ... */ };
class Z : private B { /* ... */ };
```

*public*派生使派生类成为基类的一个子类型，这是最常见的派生形式。*protected*和*private*派生用于表示实现细节。*protected*基类在经常需要进一步派生的类层次结构中非常有用，12.4.2节的*Ival_slider*是这方面的一个极好示例。*private*基类用于定义要将界面严格限制到基类定义上的那种类，这样做可以提供更强的保证。例如，指针的*Vector*模板为它的基类*Vector<void*>*增加了类型检查（13.5节）。另外，如果我们想保证对*Vec*（3.7.2节）的每个访问都将接受检查，我们就应该将*Vec*的基类声明为*private*（以防止从*Vec*到其未检查的基类*vector*的转换）

```
template<class T> class Vec : private vector<T> { /* ... */ }; // 做范围检查的向量
```

对基类的访问描述可以不写。对于这种情况，*class*基类默认为*private*，而*struct*基类则默认为*public*。例如，

```
class XX : B { /* ... */ }; // B是私有基类
struct YY : B { /* ... */ }; // B是公用基类
```

为阅读方便，最好还是始终明确写出的访问描述符。

对于基类的访问描述符控制着对基类成员的访问，以及从派生类类型到基类类型的指针和引用转换。考虑从基类*B*派生出的类*D*：

- 如果*B*是*private*基类，那么它的*public*和*protected*成员只能由*D*的成员函数和友元访问。只有*D*的成员和友元能将*D**转换到*B**。
- 如果*B*是*protected*基类，那么它的*public*和*protected*成员只能由*D*的成员函数和友元，以及由*D*派生出的类的成员函数和友元访问。只有*D*的成员和友元以及由*D*派生出的类的成员和友元能将*D**转换到*B**。
- 如果*B*是*public*基类，那么它的*public*成员可以由任何函数使用。除此之外，它的*protected*成员能由*D*的成员函数和友元，以及由*D*派生出的类的成员函数和友元访问。任何函数都能将*D**转换到*B**。

这些基本上就是成员访问规则（15.3节）的重新叙述。我们对基类访问方式的选择与对成员访问一样。例如，我前面选择让*BBwindow*作为*Ival_slider*的一个*protected*基类（12.4.2节），就是因为*BBwindow*是*Ival_slider*的实现的一部分，而不是它界面的一部分。然而我并没有把*BBwindow*作为*private*，从而将它完全隐藏起来，这是因为我还能从*Ival_slider*派生出进一步的类，而这些派生类或许还需要访问这个实现。

阐释对基类访问的技术性示例可以在C.11.2节找到。

15.3.2.1 多重继承和访问控制

在一个派生格里，如果一个名字或者基类可以从多条路径到达，那么若有一条路径使它

能够访问，它就是可访问的。例如，

```
struct B {
    int m;
    static int sm;
    // ...
};

class D1 : public virtual B { /* ... */ };
class D2 : public virtual B { /* ... */ };
class DD : public D1, private D2 { /* ... */ };

DD* pd = new DD;
B* pb = pd;           // 可以：通过D1访问
int i1 = pd->m;        // 可以：通过D1访问
```

如果一个实体可以通过多条途径到达，我们还是可能无歧义地引用它。例如，

```
class X1 : public B { /* ... */ };
class X2 : public B { /* ... */ };
class XX : public X1, public X2 { /* ... */ };

XX* pxx = new XX;
int i1 = pxx->m;       // 错误，歧义：XX::X1::B::m还是XX::X2::B::m
int i2 = pxx->sm;      // 可以：在XX里只有B::sm
```

15.3.2.2 使用声明和访问控制

不能通过使用声明取得对更多信息的访问权。使用声明只是一种能使信息的使用更方便的机制。在另一方面，如果某种访问可行，就可以将它授予其他用户。例如，

```
class B {
private:
    int a;
protected:
    int b;
public:
    int c;
};

class D : public B {
public:
    using B::a;    // 错误：B::a为私有
    using B::b;    // 可以：使B::b在整个D公开可用
};
```

将使用声明与**private**或者**protected**派生结合使用，可以用于描述一种界面，其中只将一个类所提供功能的一部分（而不是全部）呈现出来。例如，

```
class BB : private B {    // 访问 B::b 和 B::c，而不是 B::a
public:
    using B::b;
    using B::c;
};
```

另见15.2.2节。

15.4 运行时类型信息

在12.4节所定义*Ival_box*的一种可能使用方式，是将它们送给一个控制显示屏的系统，而

后，在发生了某些活动之后，系统再将对象送回应用程序。这也正是许多用户界面的工作方式。然而，用户界面系统对*Ival_box*一无所知，该系统的界面是在它自己的类和对象的基础上描述的，而不是针对我们在应用中的类。这当然是必须的，也是正确的。但是，它也确实会产生一些不良影响，因为我们会遗失掉送给系统而后又被送还我们的对象的类型。

要寻回一个对象“遗失的”类型，就要求我们能以某种方式去向对象询问其类型。针对一个对象的任何操作都要求我们有适合这个对象的指针或者引用类型。因此，检查对象类型的最明显最有用的操作是一种类型转换操作，当对象具有所期望类型时它返回一个合法的指针，如果不是这种情况就返回空指针。*dynamic_cast*运算符做的就是这件事。举个例子。假设“系统”用一个指向*BBwindow*的指针调用*my_event_handler()*，其中有动作发生。然后我就调用自己的应用代码，其中使用*Ival_box*的*do_something()*：

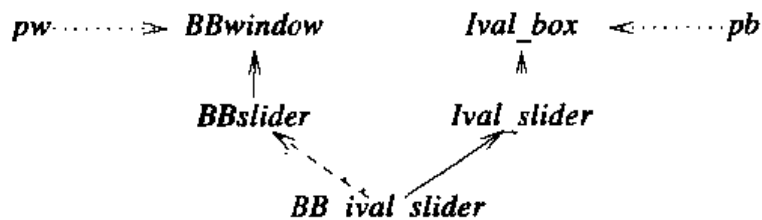
```
void my_event_handler(BBwindow* pw)
{
    if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))    // pw指向一个Ival_box?
        pb->do_something();
    else {
        // 呜呼！非预期事件
    }
}
```

解释这其中所发生的情况的一种方式：*dynamic_cast*将完成从用户界面系统的实现语言到应用所用语言的转换。最重要的是注意在这个例子里没提到的东西：对象的实际类型。这个对象应该是某种特殊的*Ival_box*，比如说*Ival_slider*；是针对某种特殊的*BBwindow*类型实现的，比如说是*BBslider*。在这种“系统”与应用的交互中，弄清对象的实际类型既无必要性，也不是我们的渴求。存在一个界面去表示交互中最本质的东西，特别是，一个设计良好的界面将隐藏起那些非本质的细节。

采用图形形式，动作

```
pb = dynamic_cast<Ival_box*>(pw)
```

可以表示为



出自*pw*和*pb*的箭头表示的是被传递的对象指针，其他箭头表示被传递对象中不同部分之间的继承关系。

在运行时对类型信息的使用一般称为“运行时类型信息”，经常简写成RTTI (Run Time Type Information)。

从基类到派生类的强制通常被称为向下强制，因为人们一般把继承树画成树根在上向下延伸。与此相对应，从派生类向基类的强制称为向上强制。从一个基类向其兄弟类的强制，例如从*BBwindow*到*Ival_box*，称为交叉强制。

15.4.1 *dynamic_cast*

运算符*dynamic_cast*有两个参数，一个是用括号“<”和“>”括起的类型，另一个是用“(”和“)”括起的指针或者引用。

首先考虑对指针的强制

dynamic_cast<*T**>(*p*)

如果*p*的类型为*T**，或为类型**D*且*T*是*D*的一个可访问的基类，结果恰如我们直接将*p*赋值给一个*T**[⊖]。例如，

```
class BB_ival_slider : public Ival_slider, protected BBslider {
    // ...
};

void f(BB_ival_slider* p)
{
    Ival_slider* pi1 = p;           // ok
    Ival_slider* pi2 = dynamic_cast<Ival_slider*>(p);           // ok

    BBslider* pbb1 = p;           // 错误: BBslider是保护的基类
    BBslider* pbb2 = dynamic_cast<BBslider*>(p);           // 可以: pbb2变成0
}
```

这是不太有趣的情况。然而这里也再次提示我们，*dynamic_cast*也不能违背对*private*和*protected*基类的保护。

*dynamic_cast*的专长是处理那些编译器无法确定转换正确性的情况。在这种情况下，

dynamic_cast<*T**>(*p*)

将查看被*p*指向的对象（如果有的话）。如果这个对象属于类*T*，或者有惟一的类型为*T*的基类，那么*dynamic_cast*就返回指向该对象的类型为*T**的指针；否则就返回0。如果*p*的值为0，*dynamic_cast*<*T**>(*p*)也返回0。注意，这里要求该转换能惟一确定一个对象。可以构造出这样的例子，使转换失败返回0，原因是被*p*所指的对象里存在着不止一个表示类型为*T*的基类的子对象（15.4.2节）。

*dynamic_cast*要求一个到多态类型的指针或者引用，以便做向下强制或者交叉强制。例如，

```
class My_slider : public Ival_slider { // 多态基类 (Ival_slider有虚函数)
    // ...
};

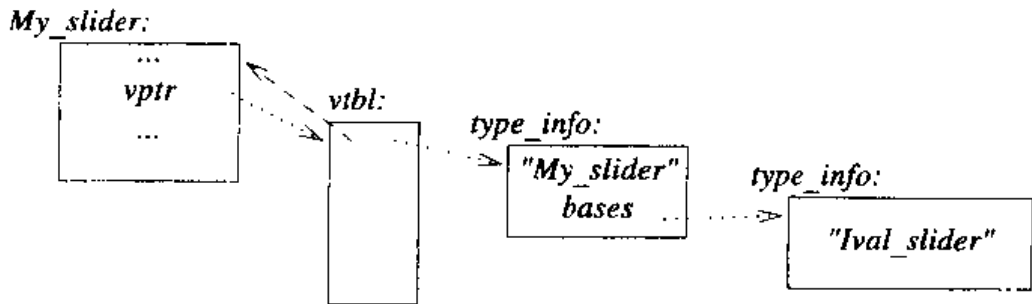
class My_date : public Date { // 基类不是多态类 (Date没有虚函数)
    // ...
};

void g(Ival_box* pb, Date* pd)
{
    My_slider* pd1 = dynamic_cast<My_slider*>(pb);           // ok
    My_date* pd2 = dynamic_cast<My_date*>(pd);           // 错误: Date不是多态类
}
```

要求指针类型为多态的，能够简化*dynamic_cast*的实现，因为这将使我们很容易为存储对象的类型信息找到一个位置。一种典型实现方式是把一个“类型信息对象”附到对象上，采用

⊖ 这里是向上强制。原书写法不清晰，这里已更正。——译者注

的方式是在这种对象的虚函数表（2.5.5节）里放一个指向类型信息的指针。例如，



图中的短划线箭头表示一个偏移量，以便在给了一个指向多态子对象的指针的情况下，能够确定整个的对象。很清楚，*dynamic_cast*确实能很有效地实现，所涉及到的全部工作不过是对代表基类的*type_info*对象做几次比较，不需要昂贵的检索或者字符串比较。

从逻辑的观点看，将*dynamic_cast*限制到多态类型上也是有意义的。理由是，如果一个对象没有虚函数，在不知道它的确切类型的情况下就无法安全地对其进行操作。因此，应该特别小心，不要把这种对象弄进了某种不能知道该对象类型的环境中。如果对象类型已知，我们就根本不需要用*dynamic_cast*。

*dynamic_cast*的目标类型不必是多态的。这就使我们可以把具体类型包裹在一个多态类型里，比如说为了将其传递穿过一个对象I/O系统（25.4.1节），而后“打开包裹”取出其中的具体类型。例如，

```

class Io_obj {           // 为对象I/O系统所用的基类
    virtual Io_obj* clone() = 0;
};

class Io_date : public Date, public Io_obj { };

void f(Io_obj* pio)
{
    Date* pd = dynamic_cast<Date*>(pio);
    // ...
}
  
```

到*void**的*dynamic_cast*可以用于确定多态类型的对象的起始地址。例如，

```

void g(Ival_box* pb, Date* pd)
{
    void* pd1 = dynamic_cast<void*>(pb);    // ok
    void* pd2 = dynamic_cast<void*>(pd);    // 错误: Date不是多态类型
}
  
```

这种技术只对与很低级的函数的交互有用。

15.4.1.1 引用的*dynamic_cast*

为了得到多态性行为，就必须通过指针或者引用去操作对象。在将*dynamic_cast*用于一个指针类型时，0表示操作失败。对引用而言，这样做既不可行，也不是人们所希望的。

在给出一个指针结果时，我们必须考虑结果为0的可能性，也就是说，考虑该指针并没有指向对象的情况。因此，对指针做*dynamic_cast*的结果必须显式检查。对于指针*p*，*dynamic_cast<T*>(p)*可以看成是一个提问，“*p*所指的对象的类型是*T*吗？”

而在另一方面，我们能合法地假定一个引用总引用着某个对象。因此，对引用*r*做*dynamic_cast<T&>(r)*，不是提问而是断言：“由*r*引用的对象的类型是*T*”。对于引用的

`dynamic_cast`的结果隐式地由`dynamic_cast`本身的实现去检查。如果对引用的`dynamic_cast`的操作对象不具有所需要的类型，就会抛出一个`bad_cast`异常。例如，

```
void f(Ival_box* p, Ival_box& r)
{
    if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {    // p是指向一个Ival_slider吗?
        // “是”时使用
    }
    else {
        // *p不是一个slider
    }

    Ival_slider& is = dynamic_cast<Ival_slider&>(r);    // r引用一个Ival_slider!
    // “是”时使用
}
```

在动态指针强制和动态引用强制的结果方面的差异，所反应的正是引用和指针之间的根本性差异。如果用户希望对引用的失败强制提供某种保护性的处理方式，就必须提供一个适当处理器。例如，

```
void g()
{
    try {
        f(new BB_ival_slider, *new BB_ival_slider);    // 参数作为Ival_box传递
        f(new BBdial, *new BBdial);    // 参数作为Ival_box传递
    }
    catch (bad_cast) {    // 14.10节
        // ...
    }
}
```

对`f()`的第一个调用将正常返回，而第二个调用将导致`bad_cast`异常，它又会被`g()`捕获。

显式检测0也可能——因此偶尔地就一定会——被人不经意地忘了。如果你担心这种情况，那么也可以写出一个转换函数，在失败时让它抛出异常，而不是返回0（15.8[1]）。

15.4.2 在类层次结构中漫游

在只使用单继承的情况下，类和其基类组成了一棵以某个基类为根的树。这很简单，但也常常过于束缚人了。在使用了多重继承之后，单个的根没有了。这件事本身并不会使事情复杂多少。但是，如果一个类在类层次结构中出现了多次，在引用表示这个类的一个或多个对象时，我们就必须多加小心。

很自然，我们会设法在应用允许的情况下尽可能保存层次结构简单（但也不能过于简单）。然而，一旦已经做出了一个不很平凡的层次结构，我们很快就会需要在其中漫游，以便找到某个适当的类，用它作为界面。这种需要的出现有两种变形。有时候，我们需要显式地指明某个基类的一个对象，或者某个基类的一个成员，15.2.3节和15.2.4.1节是这方面的例子。另外一些时候，我们在掌握着一个到完整对象或者子对象的指针的情况下，希望得到一个到表示基类的对象的指针或者指向派生类对象的指针。15.4节和15.4.1节是这方面的例子。

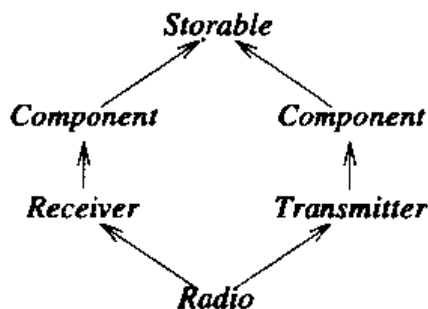
在这里，我们要考虑如何利用类型转换（强制）在类层次结构中漫游，以取得所需要的类型的指针。为了阐明在这里能使用的机制和指挥它们的规则，我们考虑包含了重复基类和虚基类的格

```

class Component : public virtual Storable { /* ... */ };
class Receiver : public Component { /* ... */ };
class Transmitter : public Component { /* ... */ };
class Radio : public Receiver, public Transmitter { /* ... */ };

```

其图形表示是



在这里，一个**Radio**对象里包含着两个类**Component**的子对象。所以，要在一个**Radio**里用 `dynamic_cast` 从 **Storable** 出发到 **Component**，就会因为歧义性而返回 `0`。因为没办法知道程序员希望的是哪个对象

```

void h1(Radio& r)
{
    Storable* ps = &r;
    // ...
    Component* pc = dynamic_cast<Component*>(ps); // pc = 0
}

```

这种歧义性在编译时一般是不可检查的

```

void h2(Storable* ps) // ps可能指向或者不指向一个Component
{
    Component* pc = dynamic_cast<Component*>(ps);
    // ...
}

```

只是需要对虚基类做这类运行时的歧义性检查。在对常规基类向下强制时（也就是说向着派生类，15.4节），总存在着对应给定强制（或者不写强制）的惟一子对象。在向上强制时（即向着基类）也存在等价的歧义性情况，这种歧义性也要在运行时捕捉。

15.4.2.1 静态和动态强制

`dynamic_cast` 能从多态性的虚基类强制到某个派生类或者兄弟类（15.4节）。`static_cast`（6.2.7节）不检查被强制的对象，所以它做不到这些：

```

void g(Radio& r)
{
    Receiver* prec = &r; // Receiver是Radio的常规基类
    Radio* pr = static_cast<Radio*>(prec); // 可以：不检查
    pr = dynamic_cast<Radio*>(prec); // 可以：运行时检查

    Storable* ps = &r; // Storable是Radio的虚基类
    pr = static_cast<Radio*>(ps); // 错误：不能从虚基类强制
    pr = dynamic_cast<Radio*>(ps); // 可以：运行时检查
}

```

*dynamic_cast*要求多态性的操作对象，这是因为在非多态对象里没有存储有关的信息，而要从一个对象出发，找到以它作为基类子对象的那些派生类对象，需要用这种信息^②。特别地，某类型的对象的布局约束条件可能是由其他语言（如C或者Fortran）决定的。对于这种类型的对象，可用的只有静态类型信息。但是另一方面，在为运行时类型识别而提供的信息中就包含了实现*dynamic_cast*所需要的信息。

为什么有人会想用*static_cast*做类层次结构的漫游呢？使用*dynamic_cast*会带来一点运行时的额外开销（15.4.1节）。更重要的是，目前存在着数以百万行计的代码是在*dynamic_cast*可以使用之前写出的。这些代码依靠其他技术来保证所做强制的合法性，在这种情况下，由*dynamic_cast*完成的检查就可能被看成是多余的。然而这些代码大都是用C风格强制写出来的（6.2.7节），常常遗留着隐秘的错误。所以，在所有可能的地方，还是应该去用更安全的*dynamic_cast*。

编译器不能对由*void**所指向的存储提供任何保证。这也意味着*dynamic_cast*不能从*void**出发进行强制，因为它必须去查看对象，以便确定其类型。对于这种情况就需要*static_cast*，例如，

```
Radio* f(void* p)
{
    Storable* ps = static_cast<Storable*>(p); // 相信程序员
    return dynamic_cast<Radio*>(ps);
}
```

*dynamic_cast*和*static_cast*都遵从*const*和访问控制规则。例如，

```
class Users : private set<Person> { /* ... */ };
void f(Users* pu, const Receiver* pcr)
{
    static_cast<set<Person*>>(pu); // 错误：访问违规
    dynamic_cast<set<Person*>>(pu); // 错误：访问违规

    static_cast<Receiver*>(pcr); // 错误：不能强制去掉const
    dynamic_cast<Receiver*>(pcr); // 错误：不能强制去掉const

    Receiver* pr = const_cast<Receiver*>(pcr); // ok
    // ...
}
```

不能强制到某个私用基类，“强制去掉*const*”（或者*volatile*）必须用*const_cast*（2.7节）。即使是那样做了，要想安全地访问强制的结果，也要求所提供的对象原本就不是用*const*（或*volatile*）声明的（10.2.7.1节）。

15.4.3 类对象的构造与析构

一个类对象并不简单地就是一块存储（4.9.6节）。类对象是通过其构造函数从“原始存储”中构筑起来的，并通过其析构函数的执行使它重归于“原始存储”。构造是一个自下而上的过程，析构则自上而下。一个类对象也就是在它得以构造或析构的意义上才称其为对象。这

② 原文不好理解，这里有所改写。问题是，你当时掌握的是某对象内部的一个代表基类的子对象的指针。当然，以它作为子对象的派生类对象可能不止一个，因为可以是多次派生（这时对象是一层套一层的）。现在想做的就是将此指针转换为指向其中某个派生类对象的指针。——译者注

种观点也反映在RTTI、异常处理（14.4.7节）和虚函数的有关规则之中。

依赖于构造或者析构的顺序是极不明智的。但是这种顺序可以在对象尚未完工之前的某点上，通过调用虚函数、*dynamic_cast*或者*typeid*（15.4.4节）的方式观察到。举个例子。如果15.4.2节类层次结构中*Component*的构造函数调用一个虚函数，它将会调用*Storable*或者*Component*中定义的版本，绝不会是*Receiver*、*Transmitter*或者*Radio*的版本。在构造之处，该对象还不是一个*Radio*，它不过是个部分构造的对象。所以，在构造或析构的过程中，最好是避免调用虚函数^①。

15.4.4 *typeid*和扩展的类型信息

*dynamic_cast*运算符能够满足运行时对于某对象的类型信息的大部分需要。最重要的是，该运算符能保证利用它写出的代码，对于由那些程序员显式提出的类所派生出的类都能正确工作。这样，*dynamic_cast*就以一种类似虚函数的方式维持了灵活性和可扩展性。

然而，知道一个对象的确切类型偶尔也会成为不可回避的问题。例如，我们或许希望知道一个类的名字或者它的布局。*typeid*运算符就是为此目的服务的，它取得一个对象，该对象代表着对应运算对象的类型。如果*typeid*()是个函数，其声明大致具有下面的样子：

```
class type_info;
const type_info& typeid(type_name) throw();           // 伪声明
const type_info& typeid(expression) throw(bad_typeid); // 伪声明
```

也就是说，*typeid*()返回一个到标准库类型*type_info*的引用，该类型在头文件<typeinfo>里定义。如果以一个*type_name*（类型名）作为运算对象，*typeid*()返回一个到*type_info*的引用，该*type_info*就代表了那个*type_name*。给一个表达式作运算对象时，*typeid*()返回一个到*type_info*的引用，该*type_info*就代表了那个表达式所指称的对象的类型。*typeid*()经常被用于找出由一个引用或者指针所引用的对象的确切类型：

```
void f(Shape& r, Shape* p)
{
    typeid(r);      // 由r引用的对象的类型
    typeid(*p);     // 由p所指的对象的类型
    typeid(p);      // 指针的类型，即Shape*（不常见，除非是写错）
}
```

如果一个多态类型的指针或者引用的操作对象的值是0，*typeid*()将抛出一个*bad_typeid*异常。如果*typeid*()的操作对象的类型不是多态的，或者它不是一个左值，其结果将在编译时确定，不需要去求值该运算对象表达式。

由实现确定的*type_info*看起来大致是下面的样子：

```
class type_info {
public:
    virtual ~type_info();           // 是多态的

    bool operator==(const type_info&) const; // 可以比较
    bool operator!=(const type_info&) const;
    bool before(const type_info&) const;     // 有序
    const char* name() const;               // 类型名
};
```

① 因为无法保证调用的就是你所需要的函数。——译者注

```
private:
    type_info(const type_info&);           // 禁止复制
    type_info& operator=(const type_info&); // 禁止复制
    // ...
};
```

函数**before()**是为了使**type_info**信息能够排序。在由**before()**定义的顺序关系和继承关系之间没有任何联系。

不能保证系统中对应于每个类型只存在着惟一的一个**type_info**对象。事实上，如果使用了动态连接库，实现将很难避免重复出现**type_info**对象的情况。因此我们总应该用 `==` 去检测**type_info**对象的相等，而不应该用 `==` 去比较这种对象的指针。

有时我们希望知道对象的确切类型，就是因为想对这个对象的整体执行某种标准服务（而不是想针对对象的某一部分）。在理想的情况下，这种服务是通过虚函数提供的，这时就不必知道对象的确切类型了。但在另一些原因下，根本无法假定对每种对象的操作能有一个公共的界面，这时就只能另辟蹊径，利用对象的确切类型就变得不可避免了（15.4.4.1节）。另一种更简单的情况就是想取得类的名字，以便产生某些诊断输出。

```
#include <typeinfo>

void g(Component* p)
{
    cout << typeid(*p).name();
}
```

类名的字符表示也由实现定义。在内存里的这种C风格字符串属于系统所有，程序员不应企图去**delete**它。

15.4.4.1 扩展的类型信息

典型地，找到一个对象的确切类型，只不过是作为获取和使用有关该类型的更详细信息的第一步。

现在考虑在运行时，一个实现或工具怎样才能将有关类型的信息提供给用户使用。比如说我有一个工具，它能对所用的每个类生成一个对象布局描述。我就可以将这种描述放进一个**map**里，使用户代码能够找到这些布局信息：

```
map <String, Layout> layout_table;

void f(B* p)
{
    Layout& x = layout_table[typeid(*p).name()];
    // 用x
}
```

另外的某些人也可能提供完全不同的信息：

```
struct TI_eq {
    bool operator()(const type_info* p, const type_info* q) { return *p==*q; }
};

struct TI_hash {
    int operator()(const type_info* p); // 计算散列值 (17.6.2.2节)
};

hash_map<type_info*, Icon, hash_fct, TI_hash, TI_eq> icon_table; // 17.6节

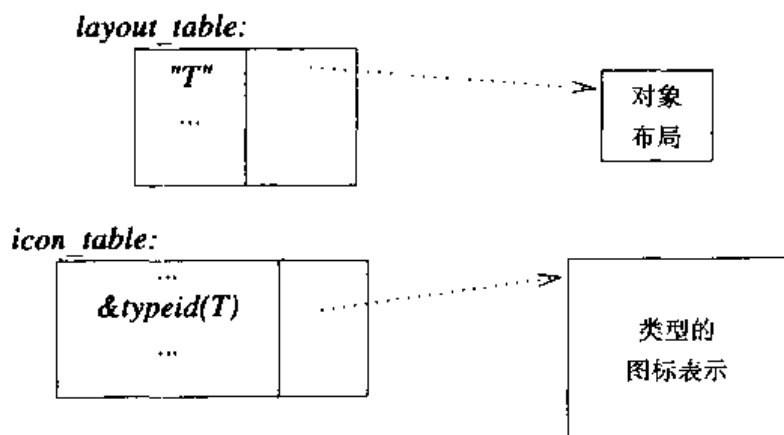
void g(B* p)
```

```

{
    Icon& i = icon_table[&typeid(*p)];
    // use i
}

```

采用这种方式建立`typeid`与信息间的关联，就使不同的人或工具可以为各种类型关联相互完全独立的不同信息：



这是非常重要的，因为想让某些人做出一组能满足所有用户需要的信息的可能性是0。

15.4.5 RTTI的使用和误用

只有在必须用的时候，才应该直接去用运行时类型信息。静态（编译时）的检查更安全，引起的开销更少，而且——在能用的地方——将导致结构更好的程序。例如，RTTI可以用于写出经过无聊伪装的`switch`语句：

```

// 运行时类型信息的误用
void rotate(const Shape& r)
{
    if (typeid(r) == typeid(Circle)) {
        // 什么也不做
    }
    else if (typeid(r) == typeid(Triangle)) {
        // 旋转三角形
    }
    else if (typeid(r) == typeid(Square)) {
        // 旋转正方形
    }
    // ...
}

```

使用`dynamic_cast`而不是`typeid`，只能使这种代码稍微有所改善。

不幸的是，这并不是稻草人式的例子，这种代码正在被人们写出来。对于由C、Pascal、Modula-2、Ada等语言训练出来的许多人来说，将程序组织为一组丑陋的`switch`语句形式几乎是无法避免的。这种冲动通常都应该压抑一下。那些需要通过类型信息完成运行时的分情况处理的地方，大部分都应该通过虚函数（2.5.5节、12.2.6节）而不是RTTI。

正确使用RTTI的许多例子大都出自下面这类地方：其中服务性的代码由一个类表示，用户又希望通过派生为它增加新功能。15.4节的`Ival_box`就是这样的例子。如果用户希望并

且能直接修改库类的定义，例如修改*BBwindow*的定义，那么就可以避免使用RTTI；如果行不通，就必须用RTTI机制。即使用户希望去修改基类，这种修改也会有其自身的问题。例如，修改时可能需要为类中的虚函数引进哑的实现，而且还是把这类虚函数引进它们根本没用也没有意义的一些类里。这类问题的一些细节在24.4.3节讨论。有关用RTTI实现一个简单对象I/O系统的例子可以在25.4.1节找到。

对于那些在依靠动态类型检查的语言（如Smalltalk或Lisp）方面有深厚基础的人，也常常会想去使用RTTI以及过于一般的类型。例如，

```
// 运行时信息的误用：
class Object { /* ... */ }; // 多态
class Container : public Object {
public:
    void put (Object*);
    Object* get ();
    // ...
};
class Ship : public Object { /* ... */ };
Ship* f(Ship* ps, Container* c)
{
    c->put(ps);
    // ...
    Object* p = c->get();
    if (Ship* q = dynamic_cast<Ship*>(p)) { // 运行时检查
        return q;
    }
    else {
        // 做另外的事（典型情况是处理错误）
    }
}
```

在这里的类*Object*就是一个完全没有必要的实现中的人为现象。它过于一般，因为它并不对应于应用领域中的任何抽象，还迫使应用程序员去使用一个实现层的抽象。解决这类问题的更好方式是使用容器模板，在其中只保存某一类指针：

```
Ship* f(Ship* ps, list<Ship*>&c)
{
    c.push_front(ps);
    // ...
    return c.pop_front();
}
```

通过与虚函数的互相配合使用，这种技术可以处理绝大部分情况。

15.5 指向成员的指针

许多类提供了简单的非常具有一般性的界面，意图是能以多种不同方式调用。举例来说，许多“面向对象”的用户界面定义了一组请求，在屏幕中出现的每个对象都要准备响应它们。此外，程序还可以直接或间接地提出这种请求。考虑这种思想的一个简单变形

```
class Std_interface {
public:
```

```

virtual void start() = 0;
virtual void suspend() = 0;
virtual void resume() = 0;
virtual void quit() = 0;
virtual void full_size() = 0;
virtual void small() = 0;

virtual ~Std_interface() {}
};

```

每个操作的确切意义将由被调用的对象定义。通常，在人或者提出请求的程序与接收请求的对象之间有一个软件层。按理想的方式，这种软件中间层不必了解各个操作（例如，`resume()` 或者 `full_size()`）的任何情况——不然的话，每当这些操作变动时，这个中间层也就必须更新了。这样，这种中间层也就是简单地从请求的发源处，将一些表示所需调用的操作的数据传递到接受方。

完成这件事的一种最简单的方式就是传送一个代表被调用操作的 *string*。例如，为调用 `suspend()`，我们可以传递字符串 `"suspend"`。然而，这样就需要在某些地方创建这种串，在其他地方进行解码，以确定它们所对应的操作——如果有的话。这看起来常常过于间接和麻烦。换一种方式，我们也可以简单地传递一个表示操作的整数。例如可以用 `2` 表示 `suspend()`。然而，虽然计算机处理整数很方便，这些整数对人却非常不方便。我们还是需要写代码，确定 `2` 意味着 `suspend()`，并进而去调用 `suspend()`。

C++ 提供了一种能间接引用类成员的功能。指向成员的指针就是一种标识类成员的值。你可以将它想像成位于该类的一个对象里的那个成员的位置，当然，实现需要负责去处理数据成员、虚函数、非虚函数等之间的差异。

考虑 `Std_interface`，如果我需要对某个对象调用 `suspend()`，而又不直接提出 `suspend()`，我就需要有一个引用着 `Std_interface::suspend()` 的指向成员的指针。我还需要有打算挂起（`suspend`）的那个对象的指针或者引用。考虑一个简单示例

```

typedef void (Std_interface::* Pstd_mem)(); // 指向成员的指针的类型
void f(Std_interface* p)
{
    Pstd_mem s = &Std_interface::suspend;
    p->suspend(); // 直接调用
    (p->*s)(); // 通过指向成员的指针调用
}

```

将取地址运算符应用到完全限定的类成员名，就能得到指向成员的指针，例如 `&Std_interface::suspend`。要声明“指向类 *X* 的成员的指针”类型的变量，需要使用形式为 `X::*` 的声明符。

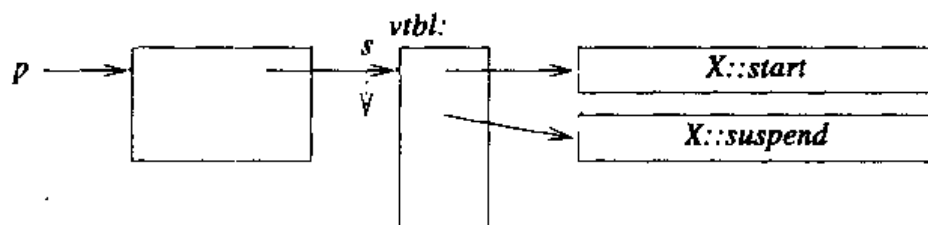
利用 `typedef` 缓和 C 声明符语法的难以阅读的缺陷，这种做法也很常见。不过还是应该注意到，`X::*` 声明符的语法形式与传统的 `*` 之间有着直接对应。

指向成员 *m* 的指针可以与一个对象组合使用，运算符 `->*` 和 `.*` 使程序员可以描述这种组合。例如，`p->*m` 将 *m* 约束于被 *p* 所指的物体，`obj.*m` 将 *m* 约束于对象 *obj*，这样得到的结果可以依照 *m* 的类型去使用。但是，不能将 `->*` 和 `.*` 的运算结果存储起来留到以后使用。

很自然，如果我们知道自己想调用的是哪个成员，我们当然会直接去调用它，不会去与这种指向成员的指针纠缠。与常规的指向函数的指针类似，在我们需要引用一个成员函数而

又不知道它的名字时，就可以采用指向成员函数的指针。然而，一个指向成员的指针并不像指向一个变量或者指向一个函数的指针，它并不是一个指向一块内存的指针。这种指针更像是到一个结构里的偏移量，或者到一个数组里的下标。当一个指向成员的指针与一个类型正确的对象的指针相结合时，它就会产生出某种东西，结果正等价于这个特定对象里的特定成员。

这些情况可以用下面的图形表示：



因为指向虚函数的指针（这个例子里的 *s*）就是某种偏移量，它并不依赖于某个对象在内存中的地址。所以，只要在两个地址空间中采用同样的对象布局，就可以在它们之间安全地传递这种指向虚成员函数的指针。然而，与指向常规函数的指针一样，指向非虚成员函数的指针不能在不同的地址空间之间安全传递。

注意，通过指向函数的指针调用的函数也可以是 *virtual*。例如，当我们通过一个指向函数的指针调用 *suspend()* 时，我们也可以为这个函数指针所应用的对象取得正确的 *suspend()*。这是对函数指针至关重要的一个方面。

一个解释器可以通过指向成员的指针去调用由字符串给出的函数

```

map<string, Std_interface*> variable;
map<string, Pstd_mem> operation;

void call_member(string var, string oper)
{
    (variable[var] -> *operation[oper])(); // var.oper()
}
  
```

在 *mem_fun()* 里可以看到指向成员的指针的一种关键使用（3.8.5节、18.4节）。

静态成员不与任何对象相关联，因此指向静态成员的指针也就是一个常规指针。例如，

```

class Task {
    // ...
    static void schedule();
};

void (*p)() = &Task::schedule; // ok
void (Task::*pm)() = &Task::schedule; // 错误：用常规指针给指向成员的指针赋值
  
```

指向数据成员的指针将在C.12节讨论。

15.5.1 基类和派生类

一个派生类至少包含由它的基类继承来的成员。通常它有更多成员。这也意味着我们可以安全地将一个指向基类成员的指针赋值给一个指向派生类的成员的指针，但反过来就不行。这种性质通常被称做反变（*contravariance*）。例如，

```

class text : public Std_interface {
public:
    void start();
    void suspend();
}
  
```

```

    // ...
    virtual void print();
private:
    vector s;
};

void (Std_interface::* pmi)() = &text::print; // 错误
void (text::* pmi)() = &Std_interface::start; // ok

```

这一反变规则看起来正好与我们关于允许用指向派生类的指针给指向基类的指针赋值的规则相反。事实上，这两个规则的存在都是为了维护一种基本保证，如果一个对象所提供的性质少于某个指针的允诺的话，该指针决不能指向这个对象。在上面情况里，`Std_interface::*` 可以应用到任何 `Std_interface`，而大部分这种对象的类型大概不是 `text`。因此，它们根本就没有成员 `text::print`，而这就是我们用于初始化 `pmi` 的成员函数。通过拒绝这种初始化，编译器就能使我们免受运行时错误之苦。

15.6 自由存储

我们可以重新定义 `operator new()` 和 `operator delete()`，通过这种方式接办存储管理工作（6.2.6.2节）。然而，取代全局的 `operator new()` 和 `operator delete()` 绝不是谨慎者们愿意做的事情。至少还有别的什么人可能依赖于默认行为的某些方面，或者也可能为这些函数提供了其他版本。

一种更可取也更好的方式是给某个特定的类提供这些操作。这个类也可以是许多派生类的基类。举个例子，我们可能希望取自12.2.6节的 `Employee` 类能够为它本身及其子类提供某种特殊的分配器和释放器：

```

class Employee {
    // ...
public:
    // ...
    void* operator new(size_t);
    void operator delete(void*, size_t);
};

```

成员 `operator new()` 和 `operator delete()` 默认为 `static` 成员。因此它们没有 `this` 指针，也不会修改任何对象。它们将提供一些存储，供构造函数去初始化，而后由析构函数去清理：

```

void* Employee::operator new(size_t s)
{
    // 分配s字节的内存，返回一个到它的指针
}

void Employee::operator delete(void* p, size_t s)
{
    // 假定p指向s个字节的由Employee::operator new()分配的存储，
    // 释放这些存储以便重用
}

```

以前使人感到神秘的 `size_t` 参数，现在在这里已经很明确了，它就被实际分配或删除的对象的大小。在删除“普通的” `Employee` 时给的参数值是 `sizeof(Employee)`，删除 `Manager` 时给的参数值是 `sizeof(Manager)`。这就使针对某个类的专用分配器不必在每次分配时保存有关对象大小的信息。自然，这种类专用的分配器也可以存储这种信息（就像通用分配器所必须做的那样），并忽略提供给 `operator delete()` 的 `size_t` 参数。然而，要在速度或者存储消耗方面对通用分配器做出重大改进是很困难的事情。

编译器怎么能知道如何给`operator delete()`提供正确的大小信息呢？如果在`delete`操作中描述的大小与对象的实际类型相匹配，问题当然很容易。然而，事情并不总是这样：

```
class Manager : public Employee {
    int level;
    // ...
};

void f()
{
    Employee* p = new Manager; // 麻烦（确切类型丢了）
    delete p;
}
```

在这种情况下，编译器将无法得到正确的大小。就像删除数组的情况一样，用户必须提供一些帮助。这个问题可以通过为基类`Employee`加一个虚析构函数的方式解决：

```
class Employee {
public:
    void* operator new(size_t);
    void operator delete(void*, size_t);
    virtual ~Employee();
    // ...
};
```

甚至是空的析构函数也可以

```
Employee::~~Employee() { }
```

从原则上说，释放工作是在析构函数（它知道大小）内完成的。进一步说，在`Employee`里给出了析构函数，就能保证每个由它派生出的类都将提供一个析构函数（这样就能保证大小正确），即使派生类里并没有用户定义的析构函数。例如，

```
void f()
{
    Employee* p = new Manager;
    delete p; // 现在一切正常（Employee是多态的）
}
```

分配工作由（编译产生的）下述调用完成

```
Employee::operator new(sizeof(Manager))
```

而释放由（编译产生的）下述调用完成

```
Employee::operator delete(p, sizeof(Manager))
```

换句话说，如果你希望自己所提供的一对分配器/释放器对派生类也能正确工作，那么或者是在基类里提供一个虚析构函数，或者就应避免在析构函数里使用`size_t`参数。自然，原来这个语言也可以设计得能从这种关注中把你拯救出来。但如果真的那样做了，也会将你“拯救”到一个不那么安全的系统里，远离各种可能的优化带来的利益。

15.6.1 数组分配

运算符`operator new()`和`operator delete()`使用户可以接办单个对象的分配和释放工作。运算符`operator new[]()`和`operator delete[]()`对于数组的分配和释放扮演着同样的角色。例如，

```

class Employee {
public:
    void* operator new[] (size_t);
    void operator delete[] (void*);
    // ...
};

void f(int s)
{
    Employee* p = new Employee[s];
    // ...
    delete[] p;
}

```

在这里，所需要的内存通过下述调用获得

```
Employee::operator new[] (sizeof(Employee)*s+delta)
```

其中的`delta`是某个由实现确定的最小开销，释放由调用

```
Employee::operator delete[] (p); // 释放 s * sizeof(Employee)+delta 字节
```

完成，元素的数目`s`以及`delta`由系统“记住”。如果实际采用的是两参数形式`delete[] ()`声明，而不是用只有一个参数的形式，那么就应该以`s*sizeof(Employee) + delta`作为第二个参数去调用它。

15.6.2 虚构造函数

在听说了虚析构造函数之后，一个明显的问题就是，“构造函数可以是虚的吗？”简单的回答是不行；而一个稍微长一点的答案是，不行，但是你很容易得到你所寻求的效果。

要构造一个对象，构造函数必须掌握所创建的对象的确切类型。因此，构造函数不能是虚的。进一步说，构造函数并不是一个普通的函数。特别是它需要与存储管理例程打交道，而且是以常规成员函数所没有的方式。因此，你不能有一个到构造函数的指针。

这两个限制都可以通过迂回方式绕过去，方法就是定义一个函数，由它调用构造函数并返回构造起来的对象。这是很幸运的，因为创建一个对象而又不必知道其确切类型，这常常也很有意义。*Ival_box_maker* (12.4.4节) 就是为做这种事情而特别设计的类的具体例子。在这里，我要给出这种想法的另一个不同的变形，其中通过克隆（复制）已有对象或者创建有关类型的新对象，将一个类的对象提供给用户。考虑

```

class Expr {
public:
    Expr(); // 默认构造函数
    Expr(const Expr&); // 复制构造函数

    virtual Expr* new_expr() { return new Expr(); }
    virtual Expr* clone() { return new Expr(*this); }
    // ...
};

```

在这里，因为像`new_expr()`和`clone()`这样的函数是虚函数，且它们（间接地）创建对象，所以也常常被称为“虚构造函数”——通过有意误用自然语言的意义。它们简单地利用某个构造函数去创建一个合适的对象。

派生类可以覆盖`new_expr()`和/或`clone()`去返回它们自己类型的对象：

```

class Cond : public Expr {
public:

```

```

    Cond();
    Cond(const Cond&);

    Cond* new_expr() { return new Cond(); }
    Cond* clone() { return new Cond(*this); }
    // ...
};

```

这也就意味着，给了一个**Expr**类的对象，用户就可以创建一个“恰好与它类型一样”的新对象。例如，

```

void user(Expr* p)
{
    Expr* p2 = p->new_expr();
    // ...
}

```

赋值给**p2**的是一个合适的指针，但它的类型却是未知的。

Cond::new_expr() 和 **Cond::clone()** 的返回类型为 **Cond***，而不是 **Expr***。这就使 **Cond** 可以克隆而不会丢失类型信息。例如，

```

void user2(Cond* pc, Expr* pe)
{
    Cond* p2 = pc->clone();
    Cond* p3 = pe->clone(); // 错误
    // ...
}

```

覆盖函数的类型必须与它要去覆盖的那个函数的类型完全一样，除了返回值类型可以松动之外。在这里，如果原来的返回值类型是 **B***，那么覆盖函数的返回值类型可以是 **D***，只要 **B** 是 **D** 的一个公用基类。与此类似，返回类型 **B&** 也可以放松为 **D&**。

注意，对于参数类型的类似放松规则将导致类型违规（15.8[12]）。

15.7 忠告

- [1] 利用常规的多重继承表述特征的合并；15.2节、15.2.5节。
- [2] 利用多重继承完成实现细节与界面的分离；15.2.5节。
- [3] 用 **virtual** 基类表达在类层次结构里对某些类（不是全部类）共同的东西；15.2.5节。
- [4] 避免显式的类型转换（强制）；15.4.5节。
- [5] 在不可避免地需要漫游类层次结构的地方，使用 **dynamic_cast**；15.4.1节。
- [6] 尽量用 **dynamic_cast** 而不是 **typeid**；15.4.4节。
- [7] 尽量用 **private** 而不是 **protected**；15.3.1.1节。
- [8] 不要声明 **protected** 数据成员；15.3.1.1节。
- [9] 如果某个类定义了 **operator delete()**，它也应该有虚析构函数；15.6节；
- [10] 在构造和析构期间不要调用虚函数；15.4.3节。
- [11] 尽量少用为解析成员名而写的显式限定词，最好是在覆盖函数里用它；15.2.1节。

15.8 练习

1. (*) 写一个 **ptr_cast** 模板，它能像 **dynamic_cast** 一样工作，只不过它不返回 **0**，而是抛出

bad_cast。

2. (*2) 写一个程序，显示在一个与RTTI有关的对象的构造期间调用构造函数的序列。对析构函数做类似的工作。
3. (*3.5) 实现Reversi/Othello棋盘游戏的一个版本。各个游戏者都可以是人或计算机。将精力集中在使程序正确，（而后）再使计算机能足够聪明，值得再次与它对弈。
4. (*3) 改进15.8[3]的用户界面。
5. (*3) 定义一个带有一组可行操作的图形对象类，作为一个图形对象库的公共基类。查看一个图形库，看看那里提供了哪些操作。定义一个带有一组可行操作的数据库对象类；查看一个数据库，看看那里提供了哪些操作。定义一个图形数据库对象类，试试使用或者不使用多重继承，讨论这两种解决方案各自的相对优势。
6. (*2) 写出15.6.2节*clone()*操作的一个版本，使它能在通过参数传来的*Arena*（10.4.11节）里安放创建的对象。实现一个简单的从*Arena*派生的*Arena*类。
7. (*2) 不要看书，写下尽可能多的C++ 关键字。
8. (*2) 写出一个符合标准的C++ 程序，其中包含一段至少有十个不同的连续的关键字，其间没有标识符、运算符、标点符号等分隔它们。
9. (*2.5) 为15.2.3.1节定义的*Radio*画出一个可行的存储布局。解释其中的虚函数调用可以怎样实现。
10. (*2) 为15.2.4节定义的*Radio*画出一个可行的存储布局。解释其中的虚函数调用可以怎样实现。
11. (*3) 考虑可以如何实现*dynamic_cast*。定义和实现一个*dcast*模板，其行为就像是*dynamic_cast*，但只依赖于你所定义的函数和数据。设法保证你可以向系统里增加类，而不必修改*dcast*或者系统里已有类的定义。
12. (*2) 假定对参数的类型检查规则被放宽，按照返回值放宽的方式，使*Derived** 可以覆盖*Base**。而后写一个程序，不使用强制就可以破坏一个类的对象。描述一种对参数类型覆盖的安全放松规则。

第三部分 标准库

这一部分将描述C++标准库。这里将介绍标准库的设计，以及在它的实现中所使用的关键性技术。这里的目标是为怎样使用这个库提供一个深入认识，阐释一些具有普遍意义的设计和编程技术，展示如何按照这个库所希望的方式去对它进行扩展。

章目

- 第16章 库组织和容器
- 第17章 标准容器
- 第18章 算法和函数对象
- 第19章 迭代器和分配器
- 第20章 串
- 第21章 流
- 第22章 数值

第16章 库组织和容器

它曾是新的、独特的和简单的。

它必须成功。

——N. Nelson

标准库的设计准则——库组织——标准头文件——语言支持——容器设计——迭代器——有基类的容器——STL容器——*vector*——迭代器——元素访问——构造函数——修改操作——表操作——大小和容量——*vector<bool>*——忠告——练习

16.1 标准库的设计

哪些东西应该出现在标准库里？一种思想是，程序员应该能在库里找到所有有意思的、重要的并具有合理普遍性的类、函数、模板等。然而，这里的问题并不是“什么东西应该出现在某个库里？”而是“什么东西应该出现在标准库里？”说“所有的东西！”是对前一问题的回答的合理的一阶逼近，但却不是对后一个问题的回答。标准库应该是某种每个实现者都必须提供的，以至于每个程序员都可以依靠的东西。

C++标准库：

- [1] 提供了对一些语言特征的支持，例如，对于存储管理（6.2.6节）和运行时类型信息的支持（15.4节）。
- [2] 提供了有关实现所确定的语言方面的一些信息，例如最大的`float`值（22.2节）。
- [3] 提供了那些无法在每个系统上由语言本身做出最优实现的函数，如`sqrt()`（22.3节）和`memmove()`（19.4.6节）。
- [4] 提供了一些非基本的功能，使程序员可以为可移植性而依靠它们，例如表（17.2.2节）、映射（17.4.1节）、排序函数（18.7.1节）和I/O流（第21章）。
- [5] 提供了一个为扩展它所提供功能的基本框架。例如，使用户可以按照内部类型I/O的风格为用户定义类型提供I/O机制的规则和支持功能。
- [6] 为其他库提供一个公共的基础。

此外，标准库还提供了少量的功能——例如随机数生成器（22.7节）——简单说，就是因为做这些事情很方便，而且也很有用。

这个库的设计主要由它所扮演的后三种角色所主导，这些角色都是密切相关的。例如，可移植性是对任何特殊的库都适用的一种重要设计准则；而公共容器类型，例如表和映射，又是在分别开发的库之间提供方便通信的基础。

从设计的角度看，这里的最后一个角色特别重要，因为它帮助划定了标准库的边界，对它的功能提出了约束。例如，标准库提供了串和表的功能，如果它们不在这里，分别开发的库就只能利用内部类型相互通信。然而，这里就没有提供模式匹配和图形功能，这些功能很

明确,使用也非常广泛,但它们却很少涉足于分别开发的库之间的通信。

除非对于支持这些角色而言,一种功能在某种意义下是必需的,否则就可以将它留给标准之外的某些库。无论是好是坏,将某些东西留给标准之外,也给那些针对某一思想提出相互竞争的具体实现的库提供了一个机会。

16.1.1 设计约束

标准库所希望扮演的角色为它的设计提出了许多约束。C++标准库提供的各种功能应该设计成为:

- [1] 对于每个学生、每个专业程序员,包括其他库的构建者,从本质上说都是极其宝贵的,而且又是能负担得起的。
- [2] 被每个程序员,在与库相关的领域中的每项工作中直接或者间接地使用。
- [3] 足够高效,能够在其他库的实现中成为手工编写的函数、类或模板的真正替代品。
- [4] 或者不依赖于具体策略,或者是使用户有机会通过参数提供所需的策略。
- [5] 在数学的意义上是最基础性的。事实上,一个能为两种微弱相关的概念服务的组件,与那种专为惟一目的设计的独立组件相比,几乎总会付出额外的开销。
- [6] 对普通使用是方便而高效的,并具有合理的安全性。
- [7] 在它们所做的工作方面是完全的。标准库可以将某些重要功能留给其他的库。但如果它接下了某项任务,它就必须提供足够的功能,使个人用户或实现者都不必再为完成某些基本工作而用其他东西去取代它。
- [8] 能很好地与内部类型和操作混合使用。
- [9] 按照默认方式是类型安全的。
- [10] 支持各种被广泛接受的程序设计风格。
- [11] 可以扩展,使之能以类似于内部类型和标准库类型的处理方式,去处理用户定义类型。

举个例子。将元素的比较准则构筑在排序函数里就是无法接受的,因为对于同样数据,也可以根据不同准则去排序。这也是为什么C标准库的`qsort()`取一个比较函数为参数,而没有依赖于某种固定的东西,例如运算符`<`的原因(7.7节)。在另一方面,为了每次比较要做一次函数调用,这样引起的负担也损害了将`qsort()`作为实现其他库的构件的能力。因为,对于几乎所有的数据类型,完成这种比较面又不强加一次函数调用的额外开销是很容易的事情。

这种开销很严重吗?对于大部分情况可能并不严重。但是,这种函数调用开销也有可能主导某些算法的执行时间,从而迫使用户去寻找其他替代品。在13.4节中描述的通过模板参数提供比较准则的技术就能解决这一问题。这个例子也说明了在高效率和通用性之间的紧张关系。一个标准库不仅要完成它的工作,还需要以足够高效的方式完成,不促使用户去提供他们自己的机制。如果不是这样,更高级机制的实现者就会被迫绕过标准库,以便能去参与竞争。这当然会给实现库的人们增加很大的负担,也会使那些希望能不依赖于平台的,或者需要使用几个分别开发的库的人们的生活带来极大的麻烦。

有关“基本性”和“方便使用”的要求看起来相互冲突。前一要求完全排除了为常见使用而优化标准库的可能性。然而,那种为常见使用的需要服务的组件(虽然不基本)还是可以作为对基本功能的补充,被包含在标准库中,但不是作为替代品。对于正交性的崇拜,不

应该阻止我们设法使新手和偶然的用户都能生活得更方便些。它也不应该导致我们容许组件中留有隐晦或者危险的默认行为。

16.1.2 标准库组织

标准库的功能都定义在`std`名字空间里，用一组头文件的方式呈现。这些头文件表明了这个库的各个主要部分。由于这种情况，列出这些头文件也就给出了标准库的一个概貌，并可以作为本章和随后几章有关标准库的描述的一个导引。

本节剩下的部分是一个头文件的列表，按照功能分组，附以简单的解释，并用讨论这些功能的章节位置作为标注。这里所采用分组的选择也符合标准的组织方式。对于C++标准手册中文本的引用（例如，s.18.1节）表明有关的功能没有在这里讨论。

所有名字以字母`c`开头的标准头文件等价于C标准库中的一个头文件。每个头文件`<X.h>`在全局名字空间和名字空间`std`里定义了C标准库中一个部分，与之对应的存在着一个头文件`<cX>`，它只在名字空间`std`里定义同样的一组名字（见9.2.2节）。

容 器		
<code><vector></code>	<code>T</code> 的一维数组	16.3节
<code><list></code>	<code>T</code> 的双向链表	17.2.2节
<code><deque></code>	<code>T</code> 的双端队列	17.2.3节
<code><queue></code>	<code>T</code> 的队列	17.3.2节
<code><stack></code>	<code>T</code> 的堆栈	17.3.1节
<code><map></code>	<code>T</code> 的关联数组	17.4.1节
<code><set></code>	<code>T</code> 的集合	17.4.3节
<code><bitset></code>	布尔量的集合	17.5.3节

关联容器 `multimap`和`multiset`可以分别在`<map>`和`<set>`里找到。`priority_queue`在`<queue>`里声明。

通用功能		
<code><utility></code>	运算符和对偶	17.1.4节, 17.4.1.2节
<code><functional></code>	函数对象	18.4节
<code><memory></code>	容器用的分配器	19.4.4节
<code><ctime></code>	C风格的日期和时间	s.20.5节

头文件`<memory>`还包含`auto_ptr`模板，它主要用于指针与异常间的平滑交互（14.4.2节）。

迭 代 器		
<code><iterator></code>	迭代器和迭代器支持	第19章

迭代器提供一些机制，使标准算法能通用于标准容器和类似类型（2.7.2节、19.2.1节）。

算 法		
<code><algorithm></code>	通用算法	第18章
<code><cstdlib></code>	<code>bsearch()</code> , <code>qsort()</code>	18.11节

一个典型通用算法可以应用于任意的具有任何类型的元素的序列（3.8节、18.3节）。C标准库函数***bsearch()***和***qsort()***可应用于以任何类型为元素的内部数组，条件是元素类型没有用户定义复制构造函数和析构函数（7.7节）。

诊 断		
<exception>	异常类	14.10节
<stdexcept>	标准异常	14.10节
<cassert>	<i>assert</i> 宏	24.3.7.2节
<cerrno>	C风格的错误处理	20.4.1节

基于异常的断言在24.3.7.1节描述。

串		
<string>	T类型的串	第20章
<cctype>	字符分类	20.4.2节
<cwctype>	宽字符分类	20.4.2节
<cstring>	C风格的字符串函数	20.4.1节
<wchar>	C风格的宽字符串函数	20.4节
<cstdlib>	C风格的串函数	20.4.1节

头文件***<cstring>***里声明了***strlen()***、***strcpy()***等一族函数。***<cstdlib>***里声明了***atof()***和***atoi()***，它们用于将C风格的字符串转换到数值。

输入/输出		
<iosfwd>	I/O功能的前导声明	21.1节
<iostream>	标准 <i>iostream</i> 对象和操作	21.2.1节
<ios>	<i>iostream</i> 基类	21.2.1节
<streambuf>	流缓冲区	21.6节
<istream>	输入流模板	21.3.1节
<ostream>	输出流模板	21.2.1节
<iomanip>	操控符	21.4.6.2节
<sstream>	以串作为I/O对象的流	21.5.3节
<cctype>	字符分类函数	20.4.2节
<fstream>	以文件作为I/O对象的流	21.5.1节
<stdio>	<i>printf()</i> 族I/O功能	21.8节
<wchar>	宽字符的 <i>printf()</i> 风格I/O	21.8节

操控符是一种对象，将它们作用于流时可以操控流的状态（例如，修改浮点数输出的格式等。21.4.6节）。

本地化		
<locale>	表示文化差异	21.7节
<clocale>	表示文化差异，C风格	21.7节

一个***locale***使一些特征本地化，如日期的输出格式，用于表示现金的字符，字符串的排列准则等。这些特征体现了不同自然语言和文化之间的差异。

语言支持		
<code><limits></code>	数值范围	22.2节
<code><climits></code>	C风格的数值标量范围宏	22.2.1节
<code><float></code>	C风格的浮点数值范围宏	22.2.1节
<code><new></code>	动态存储分配	16.1.3节
<code><typeinfo></code>	运行时类型识别支持功能	15.4.1节
<code><exception></code>	异常处理支持功能	14.10节
<code><stddef></code>	C库语言支持	6.2.1节
<code><stdarg></code>	变长函数参数表	7.6节
<code><setjmp></code>	C风格的堆栈回退	s.18.7节
<code><stdlib></code>	程序终止	9.4.1.1节
<code><time></code>	系统时钟	D.4.4.1节
<code><signal></code>	C风格的信号处理	s.18.7节

`<stddef>` 头文件定义了由 `sizeof()` 的返回类型 `size_t`，指针之差的结果类型 `ptrdiff_t`（6.2.1节）和声名狼藉的 `NULL` 宏（5.1.1节）。

数 值		
<code><complex></code>	复数及其运算	22.5节
<code><valarray></code>	数值向量和运算	22.4节
<code><numeric></code>	通用数值运算	22.6节
<code><cmath></code>	标准数学函数	22.3节
<code><stdlib></code>	C风格的随机数	22.7节

由于历史的原因，`abs()`、`fabs()`和`div()`位于 `<stdlib>`里，而不是与其他数学函数一起在 `<cmath>` 里（22.3节）。

用户或者库的实现者都不允许在标准头文件里添加或者减少任何东西。试图通过在包含头文件之前定义一些宏的方式去修改头文件的内容，或者通过在其环境中写声明去改变头文件里声明的意义，都是不可接受的（9.2.3节）。任何玩弄这种游戏的程序或实现都与标准不相符，基于这种花招的程序是不可移植的。即使它们今天得以工作，同一个实现的下一个版本也可能打破它们。请避免这种花招。

为了使用标准库的功能，必须包含有关的头文件。你自己写出的相关声明不能作为一种符合标准的替代品。这里的原因是，有些实现可能做一些基于头文件包含的优化，另一些实现可能提供由头文件触发的优化过的标准库实现。一般而言，实现者可能以用户无法预期的也不需要知道的方式去使用这些标准头文件。

然而，一个程序员可以为了非标准的库或者为用户定义的类型，给出某些功能模板（例如，`swap()`，16.3.9节）的专门化。

16.1.3 语言支持

标准库里一个很小的部分是语言支持，也就是说，是一些为了程序运行而必须提供的功能，因为有些语言特征依赖这些功能。

支持运算符 `new` 和 `delete` 的库函数已经在6.2.6节、10.4.11节、14.4.4节和15.6节讨论了。它们都在 `<new>` 中给出。

运行时类型识别依赖于类 `type_info`。它已在15.4.4节讨论过，由 `<typeinfo>` 提供。

标准异常类已经在14.10节讨论过，在 `<new>`、`<typeinfo>`、`<ios>`、`<exception>` 和 `<stdexcept>` 里给出。

程序启动和终止的问题已在3.2节、9.4节和10.4.9节讨论过。

16.2 容器设计

容器就是能保存其他对象的对象。容器的例子如表、向量和关联数组。一般说，你可以向容器里添加对象，或者从中删除对象。

很自然，这种想法可以以许多不同方式提供给用户。C++ 标准库容器的设计依据的是两条准则：在各个容器的设计中提供尽可能大的自由度，而在此同时，又使各种容器能够向用户呈现出一个公共的界面。这将使容器实现能达到最佳的效率，也使用户能写出不依赖于所使用的特定容器类型的代码。

容器的设计通常只能满足上述两准则中的一条。而标准库中的容器和算法部分（通常称为STL）可以被看做是对这个问题的一个同时提供了通用性和效率的解决方案。下面几节将讨论两种传统风格的容器设计的强项和弱项，以此作为一种走向标准容器设计方案的方式。

16.2.1 专门化的容器和迭代器

要提供向量和表，最明显方式就是对它们中的每个都按照意想的使用方式给予定义：

```
template<class T> class Vector {    // 最优方式
public:
    explicit Vector(size_t n); // 初始化为保存着n个值T()的对象

    T& operator[] (size_t);    // 下标
    // ...
};

template<class T> class List { // 最优方式
public:
    class Link { /* ... */ };

    List();           // 初始化为空
    void put(T*);     // 放到当前元素之前
    T* get();         // 取当前元素
    // ...
};
```

每个类提供一组操作，对于使用而言，这些操作基本上是最理想的。而且，我们可以为各个类选择最合适的表示方式，不必操心其他容器的情况。这也使有关操作的实现基本上是最优的。应该特别指出，最常用的操作都很小，例如 `List` 的 `put()` 和 `Vector` 的 `operator[]()`，它们很容易在线化。

对于大部分容器，一种公共使用方式就是迭代式地穿过整个容器，一个接一个地查看元素。这通常可以通过定义一个适合于容器类型的迭代器类做到。（见11.5节和11.14[7]）。

然而，在容器上的用户迭代通常并不关心数据是存在一个 `List` 里，还是存在一个 `Vector` 里。在这种情况下，迭代代码应该不依赖于所用的到底是 `List` 还是 `Vector`，最理想情况是同一段代码对两种容器都能使用。

一种解决办法是定义一个迭代器类，让它提供可以在任何容器上实现的“取得下一个元素”操作。例如，

```
template<class T> class Itor { // 公共界面（抽象类；2.5.4节和12.3节）
public:
    // 返回0表明没有下一个元素

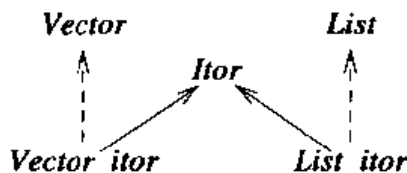
    virtual T* first() = 0; // 返回第一个元素的指针
    virtual T* next() = 0; // 返回下一个元素的指针
};
```

我们现在就可以为**Vector**和**List**提供实现了

```
template<class T> class Vector_itor : public Itor<T> { // 为Vector的实现
    Vector<T>& v;
    size_t index; // 到当前元素的下标
public:
    Vector_itor(Vector<T>& vv) : v(vv), index(0) { }
    T* first() { return (v.size()) ? &v[index=0] : 0; }
    T* next() { return (++index<v.size()) ? &v[index] : 0; }
};

template<class T> class List_itor : public Itor<T> { // 为List的实现
    List<T>& lst;
    List<T>::Link p; // 指向当前元素
public:
    List_itor(List<T>&);
    T* first();
    T* next();
};
```

或者用图形表示，短划线表示“实现使用”



这两个迭代器在内部结构上差异极大，但是这些与用户无关。我们现在可以写出能在任何容器上迭代的代码，只要已经为它实现了一个**Itor**。例如，

```
int count(Itor<char>& ii, char term)
{
    int c = 0;
    for (char* p = ii.first(); p; p = ii.next()) if (*p == term) c++;
    return c;
}
```

然而，也存在着一块暗礁。对于**Itor**迭代器的操作很简单，但却引起了一次（虚）函数调用的额外开销。在许多情况下，与要做的其他工作相比，这种额外开销是无关大局的。但是，在许多高性能的系统里，迭代穿过简单容器是最关键的操作，为实现对**Vector**或**List**的**next()**而做的一次函数调用比整数加法或者指针间接都昂贵得多。由此得出的结论是，这种模型对于标准库不合适，至少是不太理想。

无论如何，这种容器和迭代器模型已经成功地使用在许多系统中。在许多年里，它也是

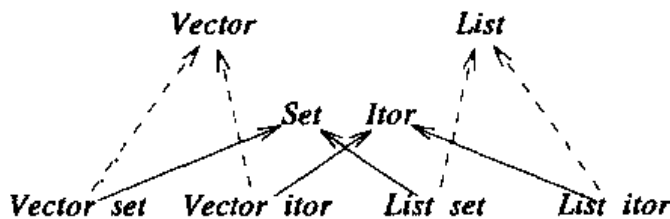
我在大部分应用中最喜欢使用的技术。关于这种技术的强项和弱项可以总结如下：

- + 个别的容器都是简单而高效的。
- + 容器间只需要很少的共性。可以通过迭代器和包装类（25.7.1节），将各种独立开发的容器纳入一个公共框架之中。
- + 使用方面的共性通过迭代器提供（而不是通过一个通用的容器类型；16.2.2节）。
- + 可以针对不同需要，为同一容器定义不同的迭代器。
- + 容器按默认方式是类型安全的、同质的（即一个容器的所有元素都属同一类型）。异质容器可以通过同质容器实现，其中保存的是指向不同类型的公共基类的指针。
- + 容器为非侵入式的（即作为容器成员的对象不必有特定基类或者指针域）。非侵入式容器可以很好地用于内部类型和具有外部规定的布局形式的`struct`。
- 每次迭代器访问都将引起一次虚函数调用的额外开销。与简单的在线访问函数相比，这种时间开销可能成为严重问题。
- 迭代器的层次结构可能变得很复杂。
- 各个容器之间不存在共同的东西，各个容器里的各个对象之间更没有共同的东西。这会使提供公共服务变得非常困难，例如提供持续性或者对象I/O。

这里的+表示优点，-表示缺点。

我认为由迭代器提供的灵活性特别重要。一个公共界面，例如`Itor`，可以在容器类（这里的`Vector`和`List`）设计和实现之后很久才提供。在我们做设计时，常常是先创造出一些相当具体的东西，例如设计了一个数组或者创建一个表。只是到后来才发现，我们需要在某种给定的环境中用某种抽象覆盖住数组和表。

实际上，我们可能会多次做这种“后续”抽象。假定我们希望表示一种集合。集合当然是与`Itor`截然不同的抽象，然而我们却可以采用与我为`Vector`和`List`提供`Itor`作为界面的同样方式，为`Vector`和`List`另外提供一个`Set`界面：



这种采用抽象类的后续抽象使我们可以为某概念提供不同的实现，实现之间可以不存在明显的类似性。例如，表和向量之间确实有一些明显的共性，但我们也很容易为`istream`实现另一个`Itor`。

从逻辑角度看，上面列表中的最后两项是本方法的主要弱点。也就是说，即使能够去掉针对容器的迭代器或其他类似界面引起的函数调用开销（这在某些环境中是可能的），这种方式对于标准库也还不太理想。

与侵入式的容器相比，非侵入式容器也会对某些容器引起一点时间和空间开销。我还没有发现这种情况成为问题。如果它真的成了问题，我们也可以为侵入式容器提供类似`Itor`这样的迭代器（16.5[11]）。

16.2.2 有基类的容器

我们也可以定义侵入式的容器，完全不必依赖于模板或者其他任何将类型声明参数化的

机制。例如，

```
struct Link {
    Link* pre;
    Link* suc;
    // ...
};

class List {
    Link* head;
    Link* curr;           // 当前元素
public:
    Link* get();          // 删除并返回当前元素
    void put(Link*);      // 在当前元素之前插入
    // ...
};
```

现在，一个`List`就是一个`Link`的表，它可以保存由`Link`派生出的任何类型的对象。例如，

```
class Ship : public Link { /* ... */ };

void f(List* lst)
{
    while (Link* po = lst->get()) {
        if (Ship* ps = dynamic_cast<Ship*>(po)) {    // Ship必须为多态类 (15.4.1节)
            // 使用ship
        }
        else {
            // 呜呼、做某些事情
        }
    }
}
```

Simula采用这种风格定义了它的标准容器，所以，这种途径可以认为是支持面向对象程序设计的语言中最早的东西。目前，所有对象的公共基类常常被称为`Object`或者别的类似名字。`Object`类除了作为容器的基本链接之外，常常还提供另外一些公共服务。

常见情况（虽然不是必须的）是，这种途径被进一步扩展，定义出一个公共容器类型：

```
class Container : public Object {
public:
    virtual Object* get();          // 删除并返回当前元素
    virtual void put(Object*);      // 在当前元素之前插入
    virtual Object*& operator[] (size_t); // 下标
    // ...
};
```

注意，由`Container`提供的操作都是虚的，所以各个容器都可以适当地覆盖它们：

```
class List : public Container {
public:
    Object* get();
    void put(Object*);
    // ...
};

class Vector : public Container {
public:
    Object*& operator[] (size_t);
    // ...
};
```

有一个问题马上就浮现出来了。我们究竟应该希望`Container`提供哪些操作？我们可以只提供每个`Container`都支持的操作。但选取所有容器操作集合的交集将形成一个很荒谬的狭窄界面。事实上，在许多有意义的情况下这个交集为空。所以，事实上，我们必须提供所希望支持的各种各样容器类基本操作集合的并集。针对一组概念的这样一个界面并集被称做一个肥大界面（24.4.3节）。

我们可以在这一肥大界面中为所有函数提供默认的实现，或者是将它们做成纯虚函数，以迫使所有派生类去实现每一个函数。在这两种情况下，我们最终都会得到许多只是简单地报告错误的函数，例如，

```
class Container : public Object {
public:
    struct Bad_op { // 异常类
        const char* p;
        Bad_op(const char* pp) : p(pp) {}
    };

    virtual void put(Object*) { throw Bad_op("put"); }
    virtual Object* get() { throw Bad_op("get"); }
    virtual Object*& operator[] (int) { throw Bad_op("{}"); }
    // ...
};
```

如果我们希望提供保护，防止某个不支持`get()`的容器制造麻烦，我们就必须在某些地方去捕捉`Container::Bad_op`。现在我们可以像下面这样写出`Ship`的例子：

```
class Ship : public Object { /* ... */ };

void f1(Container* pc)
{
    try {
        while (Object* po = pc->get()) {
            if (Ship* ps = dynamic_cast<Ship*>(po)) {
                // 使用Ship
            }
            else {
                // 呜呼！做某些事情
            }
        }
    }
    catch (Container::Bad_op& bad) {
        // 呜呼！做某些事情
    }
}
```

这是极其讨厌的，所以，最好是在其他地方做有关`Bad_op`的检查。如果依靠在其他地方捕捉异常，我们就可以将例子简化为：

```
void f2(Container* pc)
{
    while (Object* po = pc->get()) {
        Ship& s = dynamic_cast<Ship&>(*po);
        // 使用ship
    }
}
```

然而，我认为依靠运行时检查太不是滋味，也过于低效。对于这种情况，我倾向于另一种静态检查的方式：

```
void f3(Iterator<Ship>* i)
{
    while (Ship* ps = i->next()) {
        // 使用Ship
    }
}
```

有关容器设计的“有基类对象”途径，对其强项和弱项可以做出下面的总结（16.5[10]）：

- 对各个容器的操作都将引起虚函数开销。
- 所有容器必须从**Container**派生。这隐含着肥大界面的使用，要求高水平的前瞻能力，并要依靠运行时检查。要将独立开发的容器融入这个公共框架，按最好的说法，也是很难处理的事情。
- + 有了公共基类**Container**，提供了一组类似操作的容器的互换使用就很容易了。
- 容器为异质的，默认地丧失了类型安全性（我们可以依靠的仅仅是所有元素都为**Object***）。如果需要的话，可以利用模板定义出同质的类型安全的容器。
- 容器为侵入式的（即每个元素都必须来自**Object**派生出的类型）。内部类型的对象和由外界规定布局的**struct**将无法直接放入容器里。
- 从容器取出的元素，在使用之前都必须通过类型转换，以给定一个完好的类型。
- + 类**Container**和**Object**可以作为对所有容器或所有元素提供的某些服务所用的句柄。这将极大地简化提供公共服务的工作，例如持续性和对象I/O等。

如前（16.2.1节），+表示优点，—表示缺点。

与采用相互无关的容器和迭代器的途径相比，采用有基类对象的途径将不必要的复杂性推给了用户，强加了显著的运行时开销，并限制了可以放入容器的对象的种类。此外，对于许多类面言，由**Object**派生也暴露出了一个实现细节。根据这些情况，这种途径对于标准库也不理想。

当然，这种途径的普遍性和灵活性也不应该低估。与它的其他替代方案一样，这种途径也被成功使用在许多应用中。其强项在于有这样一些领域，在那里，通过单一**Container**界面所提供的简单性比效率问题更重要，以及对象I/O一类的服务方面。

16.2.3 STL容器

标准库容器和迭代器（常常被称为STL框架，3.10节）可以认为是一种能够同时取得前面描述了的两种传统模型的优点的途径，虽然这并不是STL的真实来历。STL是一心一意追求真正高效的和通用型的算法而结出的硕果。

有关效率的目标就使我们排除了采用难以在线化的虚函数去实现那些短小的、使用频繁访问的函数。因此，我们不能用抽象类作为容器的标准界面或者标准迭代器的界面，而需要让各类容器支持一个基本操作的标准集合。为了避免肥大界面问题（16.2.2节、24.4.3节），无法有效地在所有容器上实现的操作将不包括在这个公共操作集合中。例如，只为**vector**提供下标操作，但不为**list**提供。此外，为每种容器提供自己的迭代器，并让这些迭代器都支持同一组标准的迭代器操作。

标准容器不是从一个公共基类派生出来的，实际上，是每个容器实现了完整的标准容器界面。与此类似，也不存在公共的迭代器基类。这样，在使用标准容器或者迭代器时就不涉及任何显式或隐式的运行时类型检查。

最重要也是最困难的问题是如何为所有容器提供公共的服务。这个问题是通过模板参数传递的“分配器”处理的（19.4.3节），没有通过公共基类。

在进入细节和代码实例之前，我可以对STL途径的强项和弱项给出如下总结：

- + 各个容器都是简单而高效的（虽然不像真正独立的容器那么简单，但却同样高效）。
- + 每个容器都提供了一组具有同样名字和语义的标准操作。特定容器可以根据需要提供一些附加的操作。进一步说，可以用包装类（25.7.1节）将其他独立开发的容器纳入这个公共框架之中（16.5[4]）。
- + 通过标准迭代器为容器的使用提供进一步的统一性。每个容器提供了一些迭代器，它们都支持一组具有同样名字和语义的标准操作。对每个特定容器都定义了一个迭代器类型，使这些迭代器能尽可能地简单和高效。
- + 为满足对于容器的不同需要，除了标准迭代器之外，还可以定义不同的迭代器和其他通用界面。
- + 容器按照默认方式是类型安全和同质的（即，一个容器的所有元素具有统一类型）。异质容器可以通过同质容器实现，其中保存的是指向不同类型的公共基类的指针。
- + 容器为非侵入式的（即作为容器成员的对象不必具有特定基类或者指针域）。非侵入式容器可以很好地用于内部类型和具有由外部决定的布局的struct。
- + 侵入式容器也可以融入这个一般性的框架里。当然，侵入式容器将对其元素的类型强加一些限制。
- 每个容器都有一个称为`allocator`的参数，用它作为一个句柄，实现为每种容器提供的服务。这能极大地简化其他通用服务的提供，如持续性和对象I/O（19.4.3节）。
- 容器或迭代器都没有可以当做函数参数传递的标准的运行时表示形式（虽然很容易为了特定应用的需要，为标准容器和迭代器定义这样的表示形式；19.3节）。

如前（16.2.1节），+表示优点，-表示缺点。

换句话说，这些容器和迭代器并不具有固定的标准表示形式。与此相反，每个容器都以一组操作的形式提供一个标准界面，使这些容器可以互换使用。迭代器类似于句柄，这也意味着能在最小的时空开销下，使用户能利用在容器层面上（像有基类的途径那样）和迭代器层面上（像专门化容器那样）的统一性。

STL途径广泛而又深入地依靠模板机制。为避免大量的代码重复，通常需要通过专门化的方式，为保存指针的容器提供共享的实现（13.5节）。

16.3 向量

这里给出有关`vector`的描述，作为标准库容器的一个完整例子。除非另有说明，关于`vector`所说的情况都适用于每种标准库容器。第17章将描述`list`、`set`、`map`等的专有特征。这里要给出`vector`（以及类似容器）所提供的功能的一些细节，目的是使读者不但能理解`vector`的可能使用方式，还能理解它在标准库整体设计中所扮演的角色。

在17.1节里可以找到对标准容器及其所提供功能的一个综述。下面有关`vector`的介绍将分

几个步骤进行：成员类型、迭代器、元素访问、构造函数、堆栈操作、表操作、大小和容量、协助函数，以及`vector<bool>`。

16.3.1 类型

标准`vector`是定义在名字空间`std`中的一个模板，由 `<vector>` 给出。它首先定义了一组标准的类型名

```
template <class T, class A = allocator<T> > class std::vector {
public:
    // 类型:
    typedef T value_type;                // 元素类型
    typedef A allocator_type;            // 存储管理器类型
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef implementation_dependent1 iterator;    // T*
    typedef implementation_dependent2 const_iterator; // const T*
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    typedef typename A::pointer pointer;           // 元素指针
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;        // 元素引用
    typedef typename A::const_reference const_reference;
    // ...
};
```

每个标准容器都将这些类型名定义为成员。每个容器都以最适合自己的实现的方式去定义它们。

容器的元素类型作为容器的第一个模板参数传递，作为容器的`value_type`，也可以取得和使用。`allocator_type`作为模板的第二个参数，可以缺省。它定义了`value_type`与各种存储管理机制交互的方式。特别地，分配器总为容器提供了几个为其元素分配和释放存储而用的函数。分配器将在19.4节讨论。一般而言，`size_type`描述的是用做容器下标的类型，`difference_type`是求该容器的两个迭代器之差得到的结果的类型。对于大部分容器而言，它们对应于`size_t`和`ptrdiff_t`（6.2.1节）。

附录E讨论了在分配器或者元素操作抛出异常时`vector`的行为方式。

迭代器已经在2.7.2节讨论过，第19章将给出更详细的描述。可以将迭代器看做是指向容器元素的指针。每个容器都提供了一个名为`iterator`的类型，用于指向它的元素。它还提供另一个`const_iterator`类型，用于不加修改的元素访问。与指针一样，除非另有原由，我们应该总使用更安全的`const`迭代器。`vector`迭代器的实际类型是由实现定义的，对于常规定义的`vector`，最明显的定义方式就是将它们分别定义为`T*` 和`const T*`。

对`vector`的反向迭代器类型是由标准的`reverse_iterator`模板（19.2.5节）构造出来的。它们以反向顺序给出元素的序列。

正如3.8.1节所示，有了这些成员类型，用户就可以写出使用容器的代码，而不必知道所涉及的实际类型。特别是使用户可以写出能对任何标准容器工作的代码。例如，

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
```

```

typename C::const_iterator p = c.begin();    // 从头开始
while (p != c.end()) {                      // 继续到结束处
    s += *p;                                // 取得元素的值
    ++p;                                    // 使p指向下一个元素
}
return s;
}

```

必须在作为模板参数的成员类型名字之前加**typename**是件令人讨厌的事情。但无论如何，编译器不是神仙，也不存在一般有效的方式去判断一个模板参数类型的成员是不是一个类型名（C.13.5节）。

与指针一样，前缀 ***** 表示迭代器间接（2.7.2节、19.2.1节），**++** 表示迭代器增量。

16.3.2 迭代器

正如前面一节所述，迭代器可以用于在容器里漫游，而且使程序员不必知道迭代器的实际类型，就可以用它去标识容器的元素。这里有几个关键的成员函数，使程序员可以取得序列端点的元素：

```

template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 迭代器:

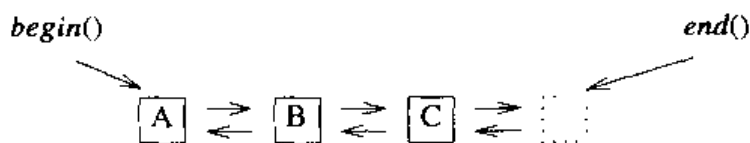
    iterator begin();                // 指向首元素
    const_iterator begin() const;
    iterator end();                  // 指向过末端一个元素位置
    const_iterator end() const;

    reverse_iterator rbegin();       // 指向反向序列的首元素
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();         // 指向反向序列的过末端一个元素位置
    const_reverse_iterator rend() const;

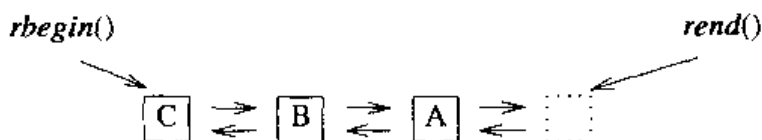
    // ...
};

```

函数对 **begin()** / **end()** 按照常规顺序给出容器的元素，即元素0，后跟元素1、元素2等。函数对 **rbegin()** / **rend()** 对按照反向序给出容器元素，即元素 $n-1$ ，而后是元素 $n-2$ 、元素 $n-3$ 等。例如，从迭代器的观点看，有如下的一个序列：



从 **reverse_iterator**（19.2.5节）看就是下面的样子：



这就使我们也可以按照反向观点去看一个序列，并对它使用算法。例如，

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
    typename C::reverse_iterator ri = find(c.rbegin(), c.rend(), v);
    if (ri == c.rend()) return c.end(); // 用c.end()表示“未发现”
    typename C::iterator i = ri.base();
    return --i;
}
```

对于`reverse_iterator`, `ri.base()`返回一个`iterator`, 指向`ri`所指之处后面的一个位置(19.2.5节)。如果没有反向迭代器, 我们将不得不写一个显式循环:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
    typename C::iterator p = c.end(); // 从末端开始收索
    while (p != c.begin())
        if (*--p == v) return p;
    return c.end(); // 用c.end()表示“未发现”
}
```

反向迭代器也是完整无缺的常规迭代器。所以我们可以写:

```
template<class C> typename C::iterator find_last(C& c, typename C::value_type v)
{
    typename C::reverse_iterator p = c.rbegin(); // 按反向观点看序列
    while (p != c.rend()) {
        if (*p == v) {
            typename C::iterator i = p.base();
            return --i;
        }
        ++p; // 注意: 增量, 不是减量(--)
    }
    return c.end(); // 用c.end()表示“未发现”
}
```

请注意, `C::reverse_iterator`与`C::iterator`不是同一个类型。

16.3.3 元素访问

与其他容器相比, `vector`的一个重要特点就是你可以方便高效地按任何顺序访问向量中的各个元素:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 元素访问
    reference operator[] (size_type n); // 不加检查的访问
    const_reference operator[] (size_type n) const;

    reference at(size_type n); // 带检查的访问
    const_reference at(size_type n) const;

    reference front(); // 首元素
    const_reference front() const;

    reference back(); // 末元素
    const_reference back() const;

    // ...
};
```

下标索引通过`operator[]()`和`at()`完成。`operator[]()`提供的是不加检查的访问，而`at()`则要作范围检查，并在下标越界时抛出`out_of_range`异常。例如，

```
void f(vector<int>& v, int i1, int i2)
try {
    for(int i = 0; i < v.size(); i++) {
        // 已做范围检查，这里用不加检查的v[i]
    }

    v.at(i1) = v.at(i2); // 访问时检查范围
    // ...
}
catch(out_of_range) {
    // 呜呼：越界错误
}
```

这也说明了使用时的一种想法。也就是说，如果已对访问范围做了检查，那么就可以安全地使用不加检查的下标运算符；否则使用带范围检查的`at()`函数是更明智的。在需要特别重视效率问题时，区分这两种情况就非常重要。如果情况不是这样，或者对于是否已经做过正确检查的情况不清楚，那么最好是采用带有检查元素范围的`[]`运算符的向量（例如3.7.2节的`Vec`类），或者采用带范围检查的迭代器（19.3节）。

默认时采用不检查的方式也是为了与数组相匹配。另外，你可以在一个快速机制之上构建起一套（带检查的）安全功能，但却无法在慢速机制上构建起一套快速功能。

访问操作返回类型为`const_reference`或者`reference`的值，具体要看它们应用的对象是不是`const`。引用常常是对元素访问最合适的类型。在对于`vector<X>`的简单而明显实现方式中，`reference`用的就是`X&`，`const_reference`用的就是`const X&`。试图创建一个越界引用的效果是无定义的。例如，

```
void f(vector<double>& v)
{
    double d = v[v.size()]; // 无定义：下标错误

    list<char> lst;
    char c = lst.front();    // 无定义：表为空
}
```

在各种标准序列里，只有`vector`和`deque`（17.2.3节）支持下标。这样做的原因是不希望通过提供从根本上低效的操作使用户感到困惑。例如，我们当然可以为`list`提供下标操作，但这样做将使效率低得吓人（是 $O(n)$ ）。

成员函数`front()`和`back()`分别返回对首元素和末元素的引用。在已知这些元素存在，且在代码里对这些元素特别有兴趣时，这些函数就很有用。将`vector`作为`stack`（16.3.5节）使用是一个最明显的例子。注意，`front()`返回对第一个元素的引用，而`begin()`返回的是到这个元素的迭代器。我通常将`front()`想像为第一个元素，而把`begin()`想像为指向第一个元素的指针。在`back()`和`end()`之间的对应就不那么简单了：`back()`是最后一个元素，而`end()`则指向最后元素的下一个位置。

16.3.4 构造函数

自然，`vector`提供了完整的一组（11.7节）构造函数、析构函数和复制操作：


```

template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 构造函数等:

    explicit vector(const A& = A());
    explicit vector(size_type n, const T& val = T(), const A& = A()); // n个val的副本
    template <class In> // In必须是输入迭代器 (19.2.1节)
        vector(In first, In last, const A& = A()); // 由 [first: last[ 复制
    vector(const vector& x);

    ~vector();

    vector& operator=(const vector& x);

    template <class In> // In必须是输入迭代器 (19.2.1节)
        void assign(In first, In last); // 由 [first: last[ 复制
    void assign(size_type n, const T& val); // n个val的副本

    // ...
};

```

vector提供了对任意元素的快速访问，但修改它的值的代价是相当昂贵的。因此，我们在创建**vector**时常常给出一个初始大小。例如，

```

vector<Record> vr(10000);

void f(int s1, int s2)
{
    vector<int> vi(s1);

    vector<double>* p = new vector<double>(s2);
}

```

这样分配的**vector**元素都将用元素类型的默认构造函数做初始化。也就是说，**vr**的10 000个元素将一个个地用**Record()**初始化，**vi**的第**s1**个元素将用**int()**初始化。请注意，内部类型的默认构造函数就是将对象初始化为适当类型的0 (4.9.5节、10.4.2节)。

如果某个类型没有默认构造函数，那么，在没有显式地为每个元素提供值的情况下，我们就不能创建以这个类型为元素的向量。例如，

```

class Num { // 无穷精度
public:
    Num(long);
    // 无默认构造函数
    // ...
};

vector<Num> v1(1000); // 错误: 无默认Num
vector<Num> v2(1000, Num(0)); // ok

```

由于一个**vector**不可能有负数个元素，所以它的大小不能是负数。这个情况也从**vector**对于**size_type**必须是**unsigned**类型的要求中反映出来。这也使得在某些系统结构中，向量的大小可以有更大范围，但是也可能产生令人吃惊的情况：

```

void f(int i)
{
    vector<char> vc0(-1); // 编译器比较容易给出警告
    vector<char> vc1(i);
}

```

```

}

void g()
{
    f(-1);    // 欺骗f()去接受一个很大正数!
}

```

在调用`f(-1)`时, `-1`将被转换为一个(很大的)正数(C.6.3节)。如果我们幸运的话, 编译器有可能找到一种方式提出警告。

`vector`的大小也可以通过初始元素的集合隐式给定。这样做就需要给构造函数提供一个值的序列, 从它们出发构造起这个`vector`。例如,

```

void f(const list<X>& lst)
{
    vector<X> v1(lst.begin(), lst.end());    // 从表复制元素

    char p[] = "despair";
    vector<char> v2(p, &p[sizeof(p)-1]);    // 从c风格串复制字符
}

```

在这两种情况, 在由其输入序列复制元素的过程中, `vector`的构造函数会调整`vector`的大小。

`vector`的一个参数的构造函数被声明为`explicit`, 以防止出人意料的转换(11.7.1节)。例如,

```

vector<int> v1(10);                // 可以: 10个int的vector
vector<int> v2 = vector<int>(10);    // 可以: 10个int的vector
vector<int> v3 = v2;                // 可以: v3是v2的副本
vector<int> v4 = 10;                // 错误: 其他将10隐式转换为vector<int>

```

复制构造函数和复制赋值运算符拷贝`vector`的所有元素。对于有许多元素的`vector`而言, 这种操作的代价可能很高, 所以`vector`通常采用引用传递。例如,

```

void f1(vector<int>&);                // 常见风格
void f2(const vector<int>&);          // 常见风格
void f3(vector<int>);                // 罕见风格

void h()
{
    vector<int> v(10000);

    // ...

    f1(v);    // 传递引用
    f2(v);    // 传递引用
    f3(v);    // 将10 000个元素复制到新向量, 以便f3()使用
}

```

在这里还有一些`assign`函数, 实现一些与多参数构造函数相对应的功能。需要这些函数, 是因为`=`的右边只能是一个运算对象, 如果要采用一个默认参数值, 或者需要提供一个范围中的一些值时, 就需要用`assign()`了。例如,

```

class Book {
    // ...
};

void f(vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10, Num(0));            // 用由10个Num(0)的副本构成的向量给vn赋值

    char s[] = "literal";
}

```

```

    vc.assign(s, &s[sizeof(s) - 1]);    // 用 "literal" 给vc赋值
    vb.assign(lb.begin(), lb.end());    // 用表的元素赋值
    // ..
}

```

因此，我们就可以用任何向量元素类型的序列对 **vector** 进行初始化，或者类似地以任何这样的序列类似地给向量赋值。重要的是，在这样做时，我们并没有引进大量的构造函数或者转换函数。请注意：赋值将完全改变一个向量里的全部元素，所有的老元素都被删除，新元素插入。在赋值之后，**vector** 的大小就是所赋值的新元素的个数。例如，

```

void f()
{
    vector<char> v(10, 'x');    // v.size() == 10, 每个元素的值都是 'x'
    v.assign(5, 'a');          // v.size() == 5, 每个元素的值都是 'a'
    // ...
}

```

当然，**assign()** 所做的事情也可以间接完成，可以首先创建一个适当的 **vector**，而后再给它赋值。例如，

```

void f2(vector<Book>& vh, list<Book>& lb)
{
    vector<Book> vt(lb.begin(), lb.end());
    vh = vt;
    // ...
}

```

但这样做既丑陋又低效。

用同样类型的两个参数创建 **vector** 会引起一种貌似歧义的情况：

```
vector<int> v(10, 50);    // vector(size, value) 或 vector(iterator1, iterator2)? vector(size, value)!
```

当然，**int** 不是迭代器，实现必须保证，在这种情况下实际调用的是：

```
vector(vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

而不是

```
vector(vector<int>::iterator, vector<int>::iterator, const vector<int>::allocator_type&);
```

库做到这一点的方式就是适当地重载构造函数，并用类似方法处理有关 **assign()** 和 **insert()** (16.3.6节) 的类似歧义问题。

16.3.5 堆栈操作

最常见的情况是，我们把 **vector** 看做一种可以直接通过下标访问元素的紧凑数据结构。因此我们可以忽略向量这个较具体的概念，而将 **vector** 看做更抽象的序列概念的一个例子。按这种观点看 **vector**，并注意到数组和 **vector** 使用中的共性，事情很清楚了，各种堆栈操作对 **vector** 也都有意义：

```

template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 堆栈操作

```

```

    void push_back(const T& x);    // 在最后加入
    void pop_back();              // 删除最后元素
    // ...
};

```

这些函数把`vector`当做一个从末端操作的堆栈。例如，

```

void f(vector<char>& s)
{
    s.push_back('a');
    s.push_back('b');
    s.push_back('c');
    s.pop_back();
    if (s[s.size()-1] != 'b') error("impossible!");
    s.pop_back();
    if (s.back() != 'a') error("should never happen!");
}

```

每次调用`push_back()`时`vector`就增长一个元素，这个元素被加在最后。所以`s[s.size()-1]`，就是`s.back()`（16.3.3节），也就是最后加进`vector`里的那个元素。

现在，除了所用的名字是`vector`而不是`stack`之外，其他都很平常。后缀`_back`用于强调这个元素是加在`vector`的末端，而不是最前面。在`vector`的末尾增加一个元素也可能成为代价高昂的操作，因为或许需要为保存它而分配额外的内存。但是，实现必须保证，由重复的堆栈操作所引起的与增长有关的开销是很少出现的。

请注意，`pop_back()`并不返回值，它只是弹出。如果我们想知道在弹出之前堆栈顶是什么，就必须自己去查看。这正好不是我最喜欢的堆栈风格（2.5.3节、2.5.4节），但一些人说这种方式更高效，而且这已是标准了。

为什么人们要在`vector`上使用类似堆栈的操作呢？一个最明显的理由就是为了实现`stack`（17.3.1节），而另一个常见原因是需要以递增的方式创建起一个`vector`。例如，我们可能需要从输入读进一个包含许多个点的`vector`，但是却不知道到底有多少个点将被读入。这样我们就无法首先分配好一个`vector`后再读入。这时就可以写

```

vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;
    while (cin >> buf) {
        if (buf == sentinel) return;
        // 检查新的点
        cities.push_back(buf);
    }
}

```

这保证了`vector`能根据需要扩展。如果我们对每个新的点要做的事情就是将它放进`vector`，我们也可以构造一个构造函数，通过输入直接对`cities`初始化（16.3.4节）。然而，对输入适当地做一些处理，并在程序工作进程中逐步扩展数据结构，这种工作方式也是很常见的。`push_back()`所支持的就是这种方式。

在C程序里，这种情况是需要使用C标准库函数`realloc()`的最常见情况之一。可见，`vector`——更一般的说法应该是所有标准容器——都能够作为`realloc()`的更通用、更优美，

而且毫不降低效率的替代机制。

`push_back()` 将隐式地导致 `vector` 的 `size()` 的增长, 所以 `vector` 不会上溢 (只要还存在可以申请到的内存, 见 19.4.1 节)。然而, `vector` 确实可能下溢:

```
void f()
{
    vector<int> v;
    v.pop_back();           // 无定义效果: v 的状态变成无定义的
    v.push_back(7);         // 无定义效果 (因为 v 的状态无定义), 可能极糟
}
```

下溢的效果是无定义的, 但 `pop_back()` 的最明显实现方式可能导致不属于 `vector` 的内存被复写。与上溢一样, 下溢也必须避免。

16.3.6 表操作

`push_back()`、`pop_back()` 和 `back()` 操作 (16.3.5 节) 使 `vector` 可以有效地作为堆栈使用。然而, 在 `vector` 的中间增加元素, 从 `vector` 里删除元素有时也很有用:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 表操作

    iterator insert(iterator pos, const T& x);           // 在 pos 前加入 x
    void insert(iterator pos, size_type n, const T& x);   // 在 pos 前加入 n 个 x 的副本
    template <class In>                                   // In 必须是输入迭代器 (19.2.1 节)
        void insert(iterator pos, In first, In last);    // 插入一批来自序列的元素

    iterator erase(iterator pos);                         // 在 pos 删除元素
    iterator erase(iterator first, iterator last);         // 删除一段元素
    void clear();                                         // 删除所有元素
    // ...
};
```

由 `insert()` 返回的迭代器指向新插入的元素。由 `erase()` 返回的迭代器指向被删除的最后元素之后的那个元素。

要看这些操作如何工作, 让我们对一个水果名字的 `vector` (随便) 做一些操作。我们首先定义一个 `vector`, 并向其中加入一些名字:

```
vector<string> fruit;
fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
```

如果我不喜欢那些名字以 `p` 开头的水果, 我就可以用如下方式删去那些名字:

```
sort(fruit.begin(), fruit.end());
vector<string>::iterator p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::iterator p2 = find_if(p1, fruit.end(), initial_not('p'));
fruit.erase(p1, p2);
```

用话来说，就是先对`vector`排序，而后找到第一个和最后一个名字以字母`p`开头的水果，最后将这些元素从`fruit`里删除。关于如何写像`initial(x)`（初始元素是`x`吗？）和`initial_not(x)`（初始元素与`x`不同吗？）一类谓词的有关问题将在18.4.2节给出解释。

操作`erase(p1, p2)`删除从`p1`开始直到`p2`的所有元素，但不包括`p2`。这可以从下面图示看出：

```

                p1          p2
                |          |
fruit[]:         v          v
              apple grape kiwifruit peach pear starfruit

```

`erase(p1, p2)` 将删除`peach`和`pear`，得到：

```

fruit[]:
      apple grape kiwifruit starfruit

```

如常，用户对序列的描述给出的是从被操作影响的序列的开始，直到超过序列结束一个位置。

有人很可能会想写

```

vector<string>::iterator p1 = find_if(fruit.begin(), fruit.end(), initial('p'));
vector<string>::reverse_iterator p2 = find_if(fruit.rbegin(), fruit.rend(), initial('p'));
fruit.erase(p1, p2+1); // 呜呼!：类型错

```

然而，`vector<fruit>::iterator`和`vector<fruit>::reverse_iterator`未必属于同一个类型，因此我们无法保证这个`erase()`能通过编译。要想与`iterator`一起使用，就必须对`reverse_iterator`做显式的转换：

```

fruit.erase(p1, p2.base()); // 从reverse_iterator提取出一个iterator (19.2.5节)

```

从`vector`里删除元素将改变它的大小，位于被删除元素之后的元素将被复制到空位中。在这个例子中，`fruit.size()`将变成4，而`star_fruit`原来位于`fruit[5]`，现在在`fruit[3]`。

很自然，我们也能删除单个的元素。在这种情况下，只需要写出索引着这个元素的迭代器（不需要一对迭代器）。例如，

```

fruit.erase(find(fruit.begin(), fruit.end(), "starfruit"));
fruit.erase(fruit.begin()+1);

```

摆脱了`starfruit`和`grape`之后，`fruit`里只剩下两个元素了

```

fruit[]:
      apple kiwifruit

```

也可以将元素插入到向量中。例如，

```

fruit.insert(fruit.begin()+1, "cherry");
fruit.insert(fruit.end(), "cranberry");

```

新元素被插入指定位置之前，从那里直到最后的元素都将被移动，以腾出空位。我们得到

```

fruit[]:
      apple cherry kiwifruit cranberry

```

注意，`f.insert(f.end(), x)`等价于`f.push_back(x)`。

我们也可以插入整个的序列

```

fruit.insert(fruit.begin()+2, citrus.begin(), citrus.end());

```

如果`citrus`是容器：

```
citrus[]:
    lemon grapefruit orange lime
```

我们将得到

```
fruit[]:
    apple cherry lemon grapefruit orange lime kiwifruit cranberry
```

`citrus`的元素被`insert()`复制进`fruit`里。而`citrus`值并不改变。

显然，`insert()`和`erase()`比那些只影响`vector`的末尾的操作（16.3.5节）更一般。但它们的代价也可能更高昂。例如，为了给新元素腾出空位，`insert()`有可能需要在内存里另一个新位置为所有元素分配空间。如果在一个容器里插入和删除一类操作非常多，或许就应该用`list`作为这个容器，而不用`vector`。`list`对于`insert()`和`erase()`是最优的，但它不适合使用下标（16.3.3节）。

在`vector`里插入删除可能导致元素被搬来搬去（对`list`或者`map`一类的关联容器都不是这样）。因此，在`insert()`和`erase()`操作之后，原本指向`vector`中某个元素的迭代器现在可能就指向了另一个元素，甚至指向的根本不是`vector`的元素^①。绝不应该再通过这种非法迭代器访问元素，这样做的结果无定义，或许会引起大灾难。特别是当心那种用于指明插入所进行的位置的迭代器。`insert()`将使它的第一个参数变为非法的。例如，

```
void duplicate_elements(vector<string>& f)
{
    for(vector<string>::iterator p = f.begin(); p != f.end(); ++p) f.insert(p, *p); // 不行!
}
```

请好好想想这个问题（16.5[15]）。`vector`的实现可能搬动所有元素——或者至少是`p`之后的所有元素——以便为新元素腾出空位。

操作`clear()`删除容器里的所有元素。这样，`c.clear()`就是`c.erase(c.begin(), c.end())`的简写形式。在`c.clear()`操作之后`c.size()`是0。

16.3.7 元素定位

最常见的情况是，`insert()`和`erase()`操作的目标是某个明确的位置（例如`begin()`或者`end()`），或者通过检索操作得到的结果（例如用`find()`），或者在迭代中确定的某个位置。在这些情况下我们都有了一个迭代器，它指明了与操作有关的元素。然而，我们常常是用下标去索引`vector`的元素。那么如果我们有了向量（或类似向量的）容器`c`的下标为7的元素，如何由它得到可以作为`insert()`和`erase()`参数的迭代器呢？因为这个元素是开始之后的第7个元素，`c.begin() + 7`就是对此的很好回答。其他看起来能对数组使用的方式都应该避免。考虑

```
template<class C> void f(C& c)
{
    c.erase(c.begin() + 7);           // 可以（如果c的迭代器支持+；参看19.2.1节）
    c.erase(&c[7]);                  // 一般不行
```

① 因为`insert()`插入元素，在原有存储无法容纳新元素时就需要重新分配存储，以存放这个`vector`里的全部元素。重新分配可能导致`vector`的全部元素被转移到新位置，这时就会释放原来的存储块，也使索引到这个存储块里的所有迭代器变得无意义（非法）了。——译者注

```

    c.erase(c+7); // 错误: 容器加7没意义
    c.erase(c.back()); // 错误: c.back() 是引用, 不是迭代器
    c.erase(c.end()-2); // 可以 (倒数第二个元素)
    c.erase(c.rbegin()+2); // 错误: vector的reverse_iterator和iterator是不同类型
    c.erase((c.rbegin()+2).base()); // 难懂, 但可行 (19.2.5节)
}

```

人们最想用的方式 `&c[7]` 对于 `vector` 的明显实现方式可以用, 在这种方式中, `c[7]` 引用相应的元素, 它的地址是合法的迭代器。但是, `c` 也可能是另一种容器, 其迭代器并不简单地就是一个指向元素的指针。例如, 对 `map` 的下标运算符 (17.4.1.3节) 返回一个 `mapped_type&`, 而不是一个到元素的引用 (其类型应该是 `value_type&`)。

并不是所有容器的迭代器都支持 `+` 运算。例如, `list` 甚至对 `c.begin()+7` 也不支持。如果你确实需要对一个 `list::iterator` 加7, 那么就只能反复地使用 `++`, 或者通过 `advance()` (19.2.1节)。

方式 `c+7` 和 `c.back()` 是简单的类型错误。容器不是数值变量, 我们不能对它加7。而 `c.back()` 是一个具有比如说 `"pear"` 一类值的元素。它并不表示容器 `c` 里的 `pear` 所在的位置。

16.3.8 大小和容量

至此, 在我们对 `vector` 的描述只最小程度地涉及到存储管理问题。`vector` 在需要时将自动增长, 通常这就是全部的事情。然而, 我们也可以直接提出 `vector` 应当如何使用存储的问题, 偶尔也有一些情况, 直接处理这个问题确实有价值。有关的操作是

```

template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // 容量操作:

    size_type size() const; // 元素个数
    bool empty() const { return size()==0; }
    size_type max_size() const; // vector的最大可能规模
    void resize(size_type sz, T val = T()); // 增加的元素用val初始化

    size_type capacity() const; // 当前已分配存储的大小 (可容纳元素个数)
    void reserve(size_type n); // 做出总数为n个元素的空位, 不初始化
    // 如果n > max_size(), 则抛出length_error

    // ...
};

```

在任意给定时刻, 一个 `vector` 里总保存着一些元素。这些元素的个数可以通过 `size()` 获得, 并可通过 `resize()` 改变。据此, 用户可以确定一个向量的大小, 在认为不够大或者过大时改变它。例如,

```

class Histogram {
    vector<int> count;
public:
    Histogram(int h) : count(max(h, 8)) {}
    void record(int i);
    // ...
};

void Histogram::record(int i)
{
    if (i<0) i = 0;
    if (count.size()<=i) count.resize(i+1); // 做出一大块空区
}

```



```
    count[i]++;
}
```

对**vector**使用**resize()**很像对于自由存储中分配的C数组使用C标准库函数**realloc()**。

为适于存放更多（或者更少）的元素而改变一个**vector**的大小时，该向量的所有元素都可能被移到了另一些新的位置。由于这种情况，把一些指向可能会改变大小的**vector**的元素的指针保存起来就绝不是个好主意了，这些指针有可能指向被释放了的内存。我们可以换个方式，保存有关的下标。请注意，**push_back()**、**insert()**和**erase()**也可能隐含着改变**vector**的大小。

除了保存元素之外，一个应用也可能为潜在的扩展而预留一些空间。了解这种扩展的程序员可以告诉**vector**的实现，要它为未来的扩展**reserve()**（预留）一些空间。例如，

```
struct Link {
    Link* next;
    Link(Link* n = 0) : next(n) {}
    // ...
};

vector<Link> v;

void chain(size_t n) // 给v填入n个Link，使每个Link指向其前一个
{
    v.reserve(n);
    v.push_back(Link(0));
    for (int i = 1; i < n; i++) v.push_back(Link(&v[i-1]));
    // ...
}
```

对**v.reserve(n)**的调用能保证，在**v**的大小增加到使**v.size()**超过**n**之前不需要再做任何存储分配。

为后面工作预留空间有两个优点。首先，即使是没有多加思考的实现也可以先分配足够的空间，以避免随着工作进展慢慢地请求足够的内存。然而，在许多情况下，由此得到的逻辑优越性可能超过效率上的收获。容器中的元素在**vector**增长时可能被重新分配位置。这样，在上面例子中，只有通过调用**reserve()**，保证在向量构建过程中不再出现分配，才能保证**v**的元素间能建立起正确的链接。也就是说，在一些情况下，**reserve()**不仅能提供效率上的优势，还能进一步提供正确性保证。

同样的保证还可以用于确保潜在的存储耗尽问题和那些代价高昂的元素重新分配只在可预期的时刻出现。对于那些有着严格的运行时间约束的程序，这些都极为重要。

请注意，**reserve()**并不改变**vector**的大小。也就是说，并没有要求它对任何新元素做初始化。在这两个方面它都与**resize()**不一样。

就像**size()**给出当时的元素个数那样，**capacity()**给出当时所保留的所有存储位置的个数。**c.capacity() - c.size()**就是在不致引起重新分配的情况下，可以插入的元素个数。

减小**vector**的大小并不会减小其容量，这样做只能为**vector**随后的增长多留下了一些空间。为将内存送还系统，需要用一点技巧^①：

① 作者在这里有些疏忽。语句执行后**v**的容量将大于或等于其元素个数（具体大小由实现确定）。无论如何**v**的元素都不会改变。另外，**v**原来的存储还保留在**tmp**中，直到**tmp**销毁时才会释放。为及早将这些内存交还系统，可以把这两个语句放进一个块里（用{和}括起）。——译者注

```
vector tmp = v; // 按照默认的容量复制v
v.swap(tmp); // 现在v具有默认的容量 (16.3.9节)
```

vector通过调用其分配器 (allocator) 的成员函数为自己的元素获得所需要的存储。默认的分配器 (称为**allocator**; 见19.4.1节) 利用**new**获取存储, 在无法得到更多存储时将抛出**bad_alloc**。其他分配器也可以采用其他策略 (19.4.2节)。

reserve() 和 **capacity()** 函数仅限于**vector**以及与它类似的紧凑型容器。**list**一类容器没有这些函数。

16.3.9 其他成员函数

许多算法——包括重要的排序算法——都涉及到元素的交换。交换元素的最明显方式 (13.5.2节) 就是简单地复制所有元素。然而, **vector**常常是用一种类似于到元素组的句柄的结构实现的 (13.5节、17.1.3节)。这样, 交换两个**vector**的工作可以通过交换它们句柄的方式更有效地实现, **vector::swap()** 做的就是这件事。在许多重要情况中, 采用这种方式与使用默认的**swap()** 相比, 在效率上可以有几个数量级的提高:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...

    void swap(vector&);

    allocator_type get_allocator() const;
};
```

函数**get_allocator()** 使程序员可以取得**vector**的分配器 (16.3.1节、16.3.4节)。如果我们希望在对应用中与某个**vector**相关的数据做分配时, 能保证采用与这个**vector**本身类似的方式 (19.4.1节), 那么就可以通过这个分配器。

16.3.10 协助函数

用 **==** 和 **<** 可以比较两个**vector**:

```
template <class T, class A>
bool std::operator==(const vector<T,A>& x, const vector<T,A>& y);

template <class T, class A>
bool std::operator<(const vector<T,A>& x, const vector<T,A>& y);
```

如果 **v1.size() == v2.size()**, 而且对每个下标 **n** 都有 **v1[n] == v2[n]** **vector v1** 和 **v2** 就相等。类似地, **<** 是按照字典顺序比较, 换句话说, 对于**vector**的 **<** 可以按如下方式定义:

```
template <class T, class A>
inline bool std::operator<(const vector<T,A>& x, const vector<T,A>& y)
{
    return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end()); // 见18.9节
}
```

也就是说, 如果在 **x[i]** 与对应 **y[i]** 的第一个不同之处 **x[i]** 小于 **y[i]**; 或者每个 **x[i]** 都等于对应的 **y[i]**, 但 **x.size() < y.size()**; 那么就认为 **x** 比 **y** 小。

标准库还提供了 **!=**、**<=**、**>** 和 **>=**, 其定义与 **==** 和 **<** 相对应。

因为**swap()** 是一个成员函数, 对它的调用应该采用 **v1.swap(v2)** 的语法形式。当然, 并不

是每个类型都有`swap`成员，所以通用型算法采用的是`swap(a, b)`的语法。为使它也能正确地处理`vector`，标准库提供了下述专门化定义：

```
template <class T, class A> void std::swap(vector<T,A>& x, vector<T,A>& y)
{
    x.swap(y);
}
```

16.3.11 `vector<bool>`

标准库还提供了专门化（13.5节）`vector(bool)`，作为`bool`类型的紧凑`vector`。`bool`变量可以寻址，所以它至少要占一个字节。当然，如果想实现一种`vector(bool)`，使每个元素只占一个二进制位也不困难。

通常的`vector`操作都可以对`vector(bool)`使用，而且保持了它们原来的意义。特别是下标和迭代器都能像所希望的那样工作。例如，

```
void f(vector<bool>& v)
{
    for (int i = 0; i < v.size(); ++i) cin >> v[i];           // 使用下标的迭代
    typedef vector<bool>::const_iterator VI;
    for (VI p = v.begin(); p != v.end(); ++p) cout << *p;    // 使用迭代器的迭代
}
```

为了做到这些，实现就必须在单个二进制位上模拟寻址操作。因为指针无法对小于一个字节的内存单元寻址，所以`vector(bool)::iterator`就可能不是指针。特别是不能认为`vector(bool)`的迭代器就是`bool*`：

```
void f(vector<bool>& v)
{
    bool* p = v.begin(); // 错误：类型不匹配
    // ...
}
```

在17.5.3节给出了一种二进制位寻址技术的梗概。

标准库还提供了布尔值集合`bitset`，带有一组布尔集合运算（17.5.3节）。

16.4 忠告

- [1] 利用标准库功能，以维持可移植性；16.1节。
- [2] 决不要另行定义标准库的功能；16.1.2节。
- [3] 决不要认为标准库比什么都好。
- [4] 在定义一种新功能时，应考虑它是否能够纳入标准库所提供的框架中；16.3节。
- [5] 记住标准库功能都定义在名字空间`std`里；16.1.2节。
- [6] 通过包含标准库头文件声明其功能，不要自己另行显示声明；16.1.2节。
- [7] 利用后续抽象的优点；16.2.1节。
- [8] 避免肥大的界面；16.2.2节。
- [9] 与自己写按照反向顺序的显式循环相比，最好是写利用反向迭代器的算法；16.3.2节。
- [10] 用`base()`从`reverse_iterator`抽取出`iterator`；16.3.2节。

- [11] 通过引用传递容器；16.3.4节。
- [12] 用迭代器类型，如`list<char>::iterator`，而不要采用索引容器元素的指针；16.3.1节。
- [13] 在不需要修改容器元素时，使用`const`迭代器；16.3.1节。
- [14] 如果希望检查访问范围，请（直接或间接）使用`at()`；16.3.3节。
- [15] 多用容器和`push_back()`或`resize()`，少用数组和`realloc()`；16.3.5节。
- [16] `vector`改变大小之后，不要使用指向其中的迭代器；16.3.8节。
- [17] 利用`reserve()`避免使迭代器非法；16.3.8节。
- [18] 在需要的时候，`reserve()`可以使执行情况更容易预期；16.3.8节。

16.5 练习

对本章中的有些练习，通过查看标准库实现的源文件可以找到有关解决方法。为你自己着想：在查看你所用的库如何解决该问题之前，先试着自己去找出一种解决方法。

1. (*1.5) 创建一个按照字母顺序保存着所有字母的`vector<char>`。按照正向和反向顺序输出这些字母。
2. (*1.5) 创建一个`vector<string>`，从`cin`为它读入一系列水果的名字，排序并输出。
3. (*1.5) 用16.5[2]的`vector`，写一个循环输出所有开头字母是a的水果名字。
4. (*1) 用16.5[2]的`vector`，写一个循环删除所有开头字母是a的水果名字。
5. (*1) 用16.5[2]的`vector`，写一个循环删除所有的柑桔类水果名字。
6. (*1.5) 用16.5[2]的`vector`，写一个循环删除所有你不喜欢的柑桔类水果名字。
7. (*2) 完成16.2.1节的`Vector`、`List`和`Itr`类。
8. (*2.5) 给出了一个`Itr`类，考虑怎样提供一批迭代器，以实现前向迭代，反向迭代，迭代通过一个在迭代过程中可能修改的容器，迭代通过在迭代中不可改变的容器。请设法组织好这组迭代器，使用户在为算法提供充分的功能时可以互换地使用这些迭代器。设法使这些容器的实现中重复的东西最少。用户还可能需要什么样的迭代器？列出你的方法的强项和弱项。
9. (*2) 完成16.2.2节的`Container`、`Vector`和`List`类。
10. (*2.5) 给了10 000个均匀分布的取值范围为0到1023的随机数，并将它们存入 a) 标准`vector`，b) 16.5[7]实现的`Vector`，c) 16.5[9]实现的`Vector`。在每种情况下计算向量中所有元素的平均值（每次都按你并不知道这个值的方式算）。统计各个循环的时间。对三种风格的向量所消耗内存的情况做出估计、度量和比较。
11. (*1.5) 写一个迭代器，使16.2.2节的`Vector`可以被当做16.2.1节风格的容器使用。
12. (*1.5) 写一个由16.2.1节的`Container`派生出的类，使16.2.2节的`Vector`可以当做一个第16.2.2节风格的容器使用。
13. (*2) 写出几个类，使16.2.1节的`Vector`和16.2.2节的`Vector`可以被当做标准容器使用。
14. (*2) 为某个现存的（非标准的，也非学生练习的）容器类型写一个模板，实现一个包含有与标准`vector`一样的成员函数和成员类型的容器。不要修改原有的容器类型。你怎样处理由原来非标准`vector`所提供的，但标准`vector`未提供的功能？
15. (*1.5) 针对一个包含三个元素`don't do this`的`vector<string>`概述16.3.6节里给出的`duplicate_elements()`的可能行为。

第17章 标准容器

是时候了，
请把你的工作置于坚实的理论基础之上。
——Sam Morgan

标准容器——容器和操作综述——效率——表示——对元素的要求——序列——*vector*
——*list*——*deque*——适配器——*stack*——*queue*——*priority_queue*——关联容器
——*map*——比较——*multimap*——*set*——*multiset*——“拟容器”——*bitset*——
数组——散列表——实现*hash_map*——忠告——练习

17.1 标准容器

标准库定义了两类容器：序列和关联容器。序列都很像*vector*（16.3节），除了专门指明的情况之外，有关*vector*所论及的成员类型和函数也都可以对任何其他容器使用，并产生同样的效果。除此之外，关联容器还提供了基于关键码访问元素的功能（3.7.4节）。

内部数组（5.2节），*string*（第20章），*valarray*（22.4节）和*bitset*（17.5.3节）中也保存元素，因此也可以看做是容器。但无论如何，这些类型都不是完整开发的标准容器。如果它们的话，就会干扰它们的基本目的。例如，内部数组不可能既能保存着它自己的大小，而且又能与C数组兼容。

有关标准容器的最关键思想，就是在所有可能之处都具有逻辑的互换性。在此之上，用户可以基于自己对效率的关心和对特定操作的需要做出选择。例如，如果经常需要基于关键码完成检索，那么就可以用*map*（17.4.1节）。换种情况，如果一般的表操作至关重要，那么就可以用*list*（17.2.2节）。如果大量添加和删除操作出现在容器的一端或两端，那么就应考虑*deque*（双端队列，17.3.2节）、*stack*（17.3.1节）或*queue*（17.3.2节）。此外，用户还可以设计其他容器，并使之可以纳入标准容器所提供的框架中（17.6节）。默认情况下应该使用*vector*（16.3节），它的实现应该对范围广泛的应用都工作得很好。

以统一方式处理不同种类的容器（以及更一般的，所有种类的信息源）的思想，引导我们走向通用型程序设计的概念（2.7.2节、3.8节）。标准库为支持这种思想提供了许多通用型算法（第18章）。这些算法可以使程序员不必再去直接处理个别容器的细节了。

17.1.1 操作综述

本节列出标准容器的公共的或者几乎公共的成员。关于更多的细节，请去读你的标准头文件（*<vector>*、*<list>*、*<map>*等，16.1.2节）。

成员类型 (16.3.1 节)	
<i>value_type</i>	元素的类型
<i>allocator_type</i>	存储管理器的类型
<i>size_type</i>	下标、元素计数等的类型
<i>difference_type</i>	迭代器之差类型
<i>iterator</i>	行为像是 <i>value_type*</i>
<i>const_iterator</i>	行为像是 <i>const value_type*</i>
<i>reverse_iterator</i>	按反向顺序查看容器, 像 <i>value_type*</i>
<i>const_reverse_iterator</i>	按反向顺序查看容器, 像 <i>const value_type*</i>
<i>reference</i>	行为像是 <i>value_type&</i>
<i>const_reference</i>	行为像是 <i>const value_type&</i>
<i>key_type</i>	关键码的类型 (仅限于关联容器)
<i>mapped_type</i>	<i>mapped_value</i> 的类型 (仅限于关联容器)
<i>key_compare</i>	比较准则的类型 (仅限于关联容器)

容器可以看成是按照该容器的 *iterator* 所定义的顺序形成的序列, 或者按反向顺序形成的序列。对于关联容器, 这个顺序则基于容器的比较准则 (默认为 $<$):

迭代器 (16.3.2 节)	
<i>begin()</i>	指向第一个元素
<i>end()</i>	指向过末端一个位置
<i>rbegin()</i>	指向按反向顺序的第一个元素
<i>rend()</i>	指向按反向顺序过末端一个位置

有些元素可以直接访问:

元素访问 (16.3.3 节)	
<i>front()</i>	第一个元素
<i>back()</i>	最后元素
<i>[]</i>	下标, 不检查 (表没有本操作)
<i>at()</i>	下标, 带检查访问 (仅对向量和双端队列有)

向量和双端队列提供了对它们的元素序列中后端元素的有效操作。表和双端队列还对它们的开始元素提供了等价的操作:

堆栈和队列操作 (16.3.5 节、17.2.2.2 节)	
<i>push_back()</i>	在最后加入
<i>pop_back()</i>	删除最后元素
<i>push_front()</i>	加入新的首元素 (仅对双端队列和表)
<i>pop_front()</i>	删除首元素 (仅对双端队列和表)

各种容器提供如下的表操作:

表操作 (16.3.6 节)	
<i>insert(p, x)</i>	在 <i>p</i> 前插入 <i>x</i>
<i>insert(p, n, x)</i>	在 <i>p</i> 前插入 <i>n</i> 个 <i>x</i>
<i>insert(p, first, last)</i>	在 <i>p</i> 前插入 <i>[first : last]</i> 的元素

表操作 (16.3.6节) (续)	
<code>erase(p)</code>	删除在 <code>p</code> 的元素
<code>erase(first, last)</code>	删除 <code>[first : last[</code>
<code>clear()</code>	删除所有元素

附录E讨论了在分配器或元素操作抛出异常时容器的行为。

所有容器都提供了与元素个数有关的各种操作和若干其他操作:

其他操作 (16.3.8节、16.3.9节、16.3.10节)	
<code>size()</code>	元素个数
<code>empty()</code>	容器为空吗?
<code>max_size()</code>	可能的最大容器的规模
<code>capacity()</code>	为 <code>vector</code> 分配的空间 (仅对向量)
<code>reserve()</code>	为后面扩充预留空间 (仅对向量)
<code>resize()</code>	改变容器的大小 (仅对向量、表和双端对列)
<code>swap()</code>	交换两个容器的所有元素
<code>get_allocator()</code>	取得容器的分配器的一个副本
<code>==</code>	两个容器的元素完全相同吗?
<code>!=</code>	两个容器的元素不同吗?
<code><</code>	一个容器按字典序在另一个之前吗?

容器提供了各种构造函数和赋值操作:

构造函数等 (16.3.4节)	
<code>container()</code>	空容器
<code>container(n)</code>	<code>n</code> 个默认值元素的容器 (关联容器没有)
<code>container(n, x)</code>	<code>n</code> 个 <code>x</code> 的拷贝 (关联容器没有)
<code>container(first, last)</code>	用 <code>[first : last[</code> 初始化元素
<code>container(x)</code>	复制构造函数, 用容器 <code>x</code> 初始化元素
<code>~container()</code>	销毁容器及其所有元素

赋值 (16.3.4节)	
<code>operator =(x)</code>	复制赋值, 元素来自容器 <code>x</code>
<code>assign(n, x)</code>	赋值 <code>n</code> 个 <code>x</code> 的拷贝 (关联容器没有)
<code>assign(first, last)</code>	用 <code>[first : last[</code> 赋值

关联容器提供如下基于关键码的检索操作:

关联操作 (17.4.1节)	
<code>operator[] (k)</code>	访问具有关键码 <code>k</code> 的元素 (对惟一关键码的容器)
<code>find(k)</code>	查找具有关键码 <code>k</code> 的元素
<code>lower_bound(k)</code>	查找具有关键码 <code>k</code> 的第一个元素
<code>upper_bound(k)</code>	查找关键码大于 <code>k</code> 的第一个元素
<code>equal_range(k)</code>	查找具有关键码 <code>k</code> 的元素的 <code>lower_bound</code> 和 <code>upper_bound</code>
<code>key_comp()</code>	关键码比较对象的副本
<code>value_comp()</code>	值比较对象的副本

除了这些公共操作外，许多容器还提供了若干特殊的操作。

17.1.2 容器综述

标准容器可以综述如下：

标准容器的操作					
	[]	表操作	前端操作	后端(堆栈)操作	迭代器
	16.3.3节	16.3.6节	17.2.2.2节	16.3.5节	19.2.1节
	17.4.1.3节	20.3.9节	20.3.9节	20.3.12节	
<i>vector</i>	const	$O(n)+$		const+	Ran
<i>list</i>		const	const	const	Bi
<i>deque</i>	const	$O(n)$	const	const	Ran
<i>stack</i>				const	
<i>queue</i>			const	const	
<i>priority_queue</i>			$O(\log(n))$	$O(\log(n))$	
<i>map</i>	$O(\log(n))$	$O(\log(n))+$			Bi
<i>multimap</i>		$O(\log(n))+$			Bi
<i>set</i>		$O(\log(n))+$			Bi
<i>multiset</i>		$O(\log(n))+$			Bi
<i>string</i>	const	$O(n)+$	$O(n)+$	const+	Ran
<i>array</i>	const				Ran
<i>valarray</i>	const				Ran
<i>bitset</i>	const				

在迭代器一列中的Ran表示随机访问迭代器，Bi表示双向迭代器。对双向迭代器的操作是随机访问迭代器操作的一个子集（19.2.1节）。表中的其他项目表示的都是操作的效率。*const*项表示这个操作所用的时间不依赖于容器中元素的数目，对常量时间的另一种习惯记法是 $O(1)$ 。 $O(n)$ 表示操作所用时间与所涉及的元素个数成比例。后缀+表明偶尔会出现显著的额外时间开销。例如，向*list*中插入元素所用时间的固定的（因此在表中列为*const*），对*vector*的同样操作则涉及到移动插入点之后的元素（所以列为 $O(n)$ ），偶尔还会出现对全部元素重新分配空间的情况（所以我加上了+）。“大O记法”是一种约定。我加上+符号是为了有助于那些除了关心平均时间还关心性能的可预见性的程序员。对于 $O(n)+$ 的规范用语是分期偿还型线性时间[⊖]。

自然，如果常量很大，它也可能超过元素个数乘以一个小的因子。但一般说，对于大量数据而言，常量时间将表示“廉价”， $O(n)$ 表示“昂贵”，而 $O(\log(n))$ 表示“相当廉价”。对于中等大小的*n*值， $O(\log(n))$ 已经更接近常量而不是 $O(n)$ 。关心操作代价的人应该注意看一看。特别是必须理解，*n*是通过计数元素的个数而得到的。在基本操作中没有“非常昂贵”的，也就是*n * n*或者更糟的操作。

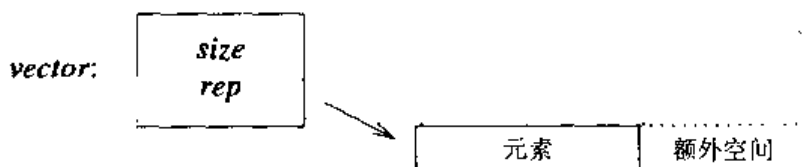
⊖ 虽然表插入有时会导致重做存储分配和元素复制，但每次操作的平均总时间仍为线性，即为 $O(n)$ ，是否重新分配并未改变复杂性。在所有操作中，典型的分期偿还型时间开销出现在对*vector*（以及*string*）的末端插入操作。该类操作一般情况下开销为常量，当需要另行分配存储时，开销将变成线性的 $O(n)$ 。 \forall const+意味着，标准库要求保证重新分配的出现较少，一系列操作的平均时间仍是常量的。——译者注

除了`string`，列在这里的代价度量所反映的都是标准的要求。`string`的度量值是我的假设。有关`stack`和`deque`的表项反映的是采用`deque`的默认实现的代价（17.3.1节、17.3.2节）。

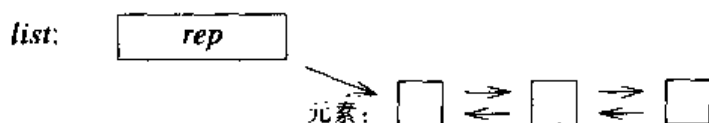
有关复杂性和代价的这些度量值都是上界。这些度量值的存在给了用户一些线索，由此可以得知自己对实现的期望。当然，实现者会设法将一些特殊情况做得更好些。

17.1.3 表示

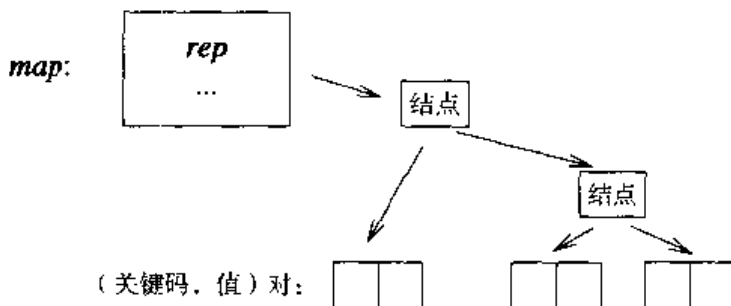
标准并没有为各种标准容器预先设定任何特定的实现方式。相反，标准只是描述了各种容器的界面，并提出了一些复杂性要求。实现者将选择适当的，一般也是经过最聪明地优化过的实现方式，以满足这些普遍性要求。容器几乎可以确定是表示为某种保存元素的数据结构，这些元素通过某种保存着大小和容量信息的句柄访问。对于`vector`，最可能的元素数据结构就像一个数组：



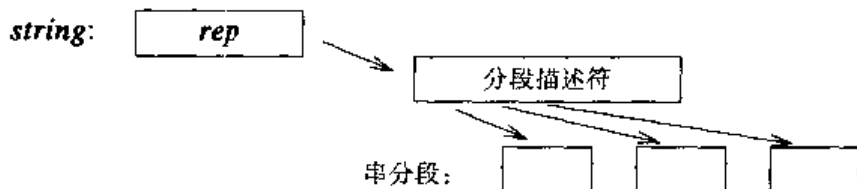
与此类似，`list`最可能就是采用一组指向的元素的连接表示：



`map`最可能被实现为结点的一棵（平衡）树，结点指向（关键码，值）对：



`string`可以按11.12节所勾勒的方式实现，或者实现为一个保存着一些字符的数组序列：



17.1.4 对元素的要求

容器里的元素是被插入对象的副本。这样，一个对象要想成为容器的元素，它就必须属于某个允许容器的实现对它进行复制的类型。容器可以利用复制构造函数或者赋值做元素的复制工作，在这两种情况下，都要求复制结果是一个等价的对象。这个要求大致意味着你在容器值上设置的所有可能的相等检测都必须认为这个副本与原来的对象相等。换句话说，复

制一个元素必须能像内部类型（包括指针）的常规复制一样工作。例如，

```
X& X::operator=(const X& a) // 完好的赋值运算符
{
    // 将a的所有成员复制到this*
    return *this;
}
```

使X可以被接受作为标准容器的元素。但是

```
void Y::operator=(const Y& a) // 不良的赋值运算符
{
    // 将a的所有成员清0
}
```

使Y不合适被接受，因为Y的赋值既没有常规的返回类型，也不具有常规的语义。

一个编译器可以检查出某些违背标准库规则的情况，但另一些则检查不出来，它们或许会导致无法预期的行为。例如，一个能够抛出异常的赋值操作可能留下一个部分复制的元素。这种情况也可能使某个元素处在一种非法状态，该状态后来可能引起麻烦。这种复制操作本身就属于不良设计（14.4.6.1节、附录E）。

当复制元素的方式不正确时，可以换一种方式，把指向对象的指针放入容器，而不是放这些对象本身。最明显的例子是多态类型（2.5.4节、12.2.6节）。例如，为了保证多态性的行为，我们前面用的就是`vector<Shape*>`而不是`vector<Shape>`。

17.1.4.1 比较

关联容器要求其元素是有序的。许多可以应用于容器的操作也是这样（例如，`sort()`）。按照默认方式，这个序由`<`运算符定义。如果`<`不合适，程序员就必须另外提供合适的比较操作（17.4.1.5节、18.2.4节）。排序准则必须定义了一种严格的弱顺序。非形式地说，这就要求小于和等于都必须是传递的。也就是说，对于顺序准则`cmp`有：

- [1] `cmp(x, x)` 是`false`。
- [2] 如果`cmp(x, y)` 且`cmp(y, z)`，那么`cmp(x, z)`。
- [3] 定义`equiv(x, y)` 为 `!(cmp(x, y) || cmp(y, x))`。如果`equiv(x, y)` 且`equiv(y, z)`，那么`equiv(x, z)`。

考虑

```
template<class Ran> void sort(Ran first, Ran last); // 用<做比较
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp); // 用cmp
```

第一个版本用`<`做比较，第二个用用户定义的`cmp`做比较。举个例子，我们可能确定用不区分大小写的方式对`fruit`排序。要做这件事，我们定义一个函数对象（11.9节、18.4节），用一对`string`调用时，它能够完成这个比较：

```
class Ncase { // 不区分大小写的比较
public:
    bool operator()(const string&, const string&) const;
};

bool Ncase::operator()(const string& x, const string& y) const
    // 若x按字典序小于y就返回true，不考虑大小写
{
    string::const_iterator p = x.begin();
    string::const_iterator q = y.begin();
```

```

while (p!=x.end() && q!=y.end() && toupper(*p)==toupper(*q)) {
    ++p;
    ++q;
}
if (p==x.end()) return q!=y.end();
if (q==y.end()) return false;
return toupper(*p) < toupper(*q);
}

```

我们用这个比较准则调用`sort()`。例如，给定

```

fruit:
apple pear Apple Pear lemon

```

用`sort(fruit.begin(), fruit.end(), Nocase())`将产生

```

fruit:
Apple apple lemon Pear pear

```

这样的情况，而简单的`sort(fruit.begin(), fruit.end())`将给出

```

fruit:
Apple Pear apple lemon pear

```

假定在所用的字符集中大写字母排在小写字母之前。

当心，如果对C风格的字符串（即`char*`）使用`<`，所定义的并不是字典序（13.5.2节）。这样，如果使用C风格字符串，关联容器将不能像大部分人所期望的那样工作。要使其正确工作，就必须用一个能完成字典序比较的小于函数。例如，

```

struct Cstring_less {
    bool operator()(const char* p, const char* q) const { return strcmp(p, q) < 0; }
};

map<char*, int, Cstring_less> m;    // 用strcmp()比较const char*的map

```

17.1.4.2 其他关系运算符

按默认规定，容器和算法在需要小于比较时都采用`<`。如果默认方式不合适，程序员就必须另行提供一个比较准则。但这里没有提供机制使人可以同时传递一个相等检测。相反，如果程序员提供了比较`cmp`，完成相等检测就需要用两次比较。例如，

```

if (x == y) // 在用户提供比较的地方不行

if (!cmp(x, y) && !cmp(y, x)) // 在用户提供比较cmp的地方能行

```

这就使我们不必为每个关联容器和几乎所有的算法再增加一个相等参数。这看起来代价高了一点，但标准库并不经常检查相等，在50%情况下只需要调用一次`cmp()`。

在实践中也经常用小于关系（默认为`<`）定义一个等价关系而不是相等（默认为`==`）。举例来说，关联容器（17.4节）比较关键码使用的就是等价检测`!(cmp(x, y) || cmp(y, x))`。这也意味着等价的关键码并不必完全一样。例如，采用不区分大小写的比较作为比较准则的`multimap`（17.4.2节），将认为`Last`、`last`、`lAst`、`laST`和`lasT`等价，虽然对于`string`的`==`认为它们不同。这也使我们能在排序时忽略那些不重要的差异。

有了`<`和`==`之后，我们很容易构造出其他常用比较。标准库在名字空间`std::rel_ops`里定义了它们，并通过`<utility>`给出：

```
template<class T> bool rel_ops::operator!=(const T& x, const T& y) { return !(x==y); }
template<class T> bool rel_ops::operator>(const T& x, const T& y) { return y<x; }
template<class T> bool rel_ops::operator<=(const T& x, const T& y) { return !(y<x); }
template<class T> bool rel_ops::operator>=(const T& x, const T& y) { return !(x<y); }
```

将这些运算符放进`rel_ops`，保证了在需要它们时很容易去用。当然，除非是将它们由名字空间里提取出来，否则它们也是不能用的：

```
void f()
{
    using namespace std;
    // 按默认不产生 !=、> 等
}

void g()
{
    using namespace std;
    using namespace std::rel_ops;
    // 默认地产生 !=、> 等
}
```

没有把 `!=` 等运算符直接定义在名字空间 `std` 里，因为并不总需要它们，而且有时它们的定义还会与用户代码相互干扰。例如，假定我写了一个通用数学库，我可能就希望用自己的关系运算，而不是标准库的版本。

17.2 序列

序列遵循 `vector`（16.3节）所描述的模式。标准库提供的基本序列包括：

`vector` `list` `deque`

从它们出发，通过定义适当的界面，生成了

`stack` `queue` `priority_queue`

这几个序列被称为容器适配器、序列适配器，或者简称适配器（17.3节）。

17.2.1 向量——`vector`

在16.3节已经描述了标准 `vector` 的细节。只有 `vector` 提供了预留空间（16.3.8节）的功能。按默认规定，用 `[]` 的下标操作不做范围检查，如果需要检查，请用 `at()`（16.3.3节）、带检查的向量（3.7.2节）或者带检查的迭代器（19.3节）。`vector` 提供随机访问迭代器（19.2.1节）。

17.2.2 表——`list`

`list` 是一种最合适于做元素插入和删除的序列。与 `vector`（以及 `deque`，17.2.3节）相比，对 `list` 做下标操作慢得令人讨厌，因此 `list` 没有提供下标操作。也是由于这种原因，`list` 提供的是双向迭代器（19.2.1节）而不是随机访问迭代器。这隐含着 `list`（典型地）是用某种形式的双向链表实现的（17.8[16]）。

`list` 给出了 `vector` 所提供的几乎所有成员类型和操作，例外的就是下标、`capacity()` 和 `reserve()`：

```
template <class T, class A = allocator<T> > class std::list {
public:
```

```

// 类型和操作如vector, 除[]、at()、capacity()和reserve()之外
// ...
};

```

17.2.2.1 粘接、排序和归并

除了一般序列的操作外, *list*还提供了若干特别适用于对表进行处理的操作:

```

template <class T, class A = allocator<T>> class list {
public:
    // ...
    // 表的特殊操作

    void splice(iterator pos, list& x);           // 将x的所有元素移到
                                                    // 本表的pos之前, 且不做复制
    void splice(iterator pos, list& x, iterator p); // 将x中的 *p移到
                                                    // 本表的pos之前, 且不做复制
    void splice(iterator pos, list& x, iterator first, iterator last);

    void merge(list&);           // 归并排序的表
    template <class Cmp> void merge(list&, Cmp);

    void sort();
    template <class Cmp> void sort(Cmp);

    // ...
};

```

这些*list*操作都是稳定的, 也就是说, 它们都能保持等价元素的相对位置不变。

16.3.6节的*fruit*实例在将*fruit*定义为*list*后同样可以工作。此外, 我们通过一个“粘接”(splice)操作, 就可以从一个表中取出一些元素, 并将它们插入另一个表里。给了

```

fruit:
    apple  pear

citrus:
    orange  grapefruit  lemon

```

我们可以像

```

list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin());

```

这样将*orange*从*citrus*粘接入*fruit*。这样做的效果是从*citrus*(*citrus.begin()*)删去了第一个元素, 并将它放到*fruit*里的第一个名字以*p*开始的元素之前, 结果是

```

fruit:
    apple  orange  pear

citrus:
    grapefruit  lemon

```

注意, *splice()*并不像*insert()*那样复制元素(16.3.6节), 它只是简单地修改引用这些元素的*list*数据结构。

除了粘接个别元素和一段区间中的元素外, 也可能*splice()*一个*list*的全部元素:

```

fruit.splice(fruit.begin(), citrus);

```

这将产生

```

fruit:
    grapefruit  lemon  apple  orange  pear

```

```
citrus:
    <empty>
```

splice() 的各个版本都以它第二个参数 *list* 作为取元素的地方, 这就使元素能够从原来的表中取下。只用迭代器做不到这一点, 因为如果仅有到某个元素的迭代器 (18.6 节), 将没有一般的方式去确定保存这个元素的容器。

很自然, 这里的迭代器参数必须是合法的迭代器, 确实指向应该指向的那个 *list*。也就是说, 它或者指向该 *list* 的一个元素, 或者指向该 *list* 的 *end()*。如果不是这样, 结果将无定义, 很可能是大灾难。例如,

```
list<string>::iterator p = find_if(fruit.begin(), fruit.end(), initial('p'));
fruit.splice(p, citrus, citrus.begin());      // ok
fruit.splice(p, citrus, fruit.begin());        // 错误: fruit.begin() 并不指向 citrus
citrus.splice(p, fruit, fruit.begin());        // 错误: p 不指向 citrus
```

即使 *citrus* 为空表, 第一个 *splice()* 也没问题。

merge() 组合起两个排好序的表, 方式是将一个 *list* 的元素都取出来, 将它们都放入另一个表, 且维持正确的顺序。例如,

```
f1:
    apple quince pear
f2:
    lemon grapefruit orange lime
```

可以按

```
f1.sort();
f2.sort();
f1.merge(f2);
```

这样的方式排序和归并, 结果是

```
f1:
    apple grapefruit lemon lime orange pear quince
f2:
    <empty>
```

如果有一个表未排序, *merge()* 仍然能产生出一个表, 其中包含着原来两个表的元素的并集。当然, 对结果的顺序就没有任何保证了。

像 *splice()* 一样, *merge()* 也不复制元素。它只是从源表中取出元素, 再将它们粘接到目标表里。在 *x.merge(y)* 之后, *y* 将变成空表。

17.2.2.2 前端操作

list 也提供一些针对第一个元素的操作, 与每个容器都提供的针对最后元素的操作 (16.3.6 节) 相对应:

```
template <class T, class A = allocator<T> > class list {
public:
    // ...
    // 元素访问

    reference front();           // 引用第一个元素
    const_reference front() const;

    void push_front(const T&);    // 加入新的第一个元素
```

```

void pop_front();           // 删除第一个元素
// ...
);

```

容器的第一个元素称做它的`front`。对`list`而言，前端操作与后端操作（16.3.5节）一样方便而高效。如果可以选择的话，最好是用后端操作而不是前端操作。采用后端操作写出的代码不但可以对`list`使用，也能对`vector`使用。所以，如果有机会，使针对`list`写出的代码最终演变为应用于各种容器的通用型算法，那么最好就是采用具有更广泛可用性的后端操作。这是得到最大灵活性的规则的一种极特殊情况。在完成一个工作时，只使用最小的基本操作集合通常是更明智的（17.1.4.1节）。

17.2.2.3 其他操作

对于`list`的插入和删除操作效率特别高。当这类操作非常频繁时，这种性质就会使人们倾向于使用`list`。这反过来又使直接支持某些删除元素的通用方法变得很有价值：

```

template <class T, class A = allocator<T> > class list (
public:
    // ...

    void remove(const T& val);
    template <class Pred> void remove_if(Pred p);

    void unique();                               // 根据 == 删除重复元素
    template <class BinPred> void unique(BinPred b); // 根据b删除重复元素

    void reverse();                             // 元素反转
);

```

例如，给了

```

fruit:
apple orange grapefruit lemon orange lime pear quince

```

我们可以像

```
fruit.remove("orange");
```

这样删除所有值为 "orange" 的元素，结果是

```

fruit:
apple grapefruit lemon lime pear quince

```

通常，人们最感兴趣的一般不是简单地删除所有元素，而是删除满足某种准则的所有元素。`remove_if()` 操作完成的的就是这件事。例如，

```
fruit.remove_if(initial('l'));
```

将从`fruit`里删除所有名字的开始字母为 'l' 的所有元素，给出的是

```

fruit:
apple grapefruit pear quince

```

删除元素的另一个常见原因是希望去掉重复，`unique()` 操作就是为此而提供的。例如，

```

fruit.sort();
fruit.unique();

```

先排序的原因是`unique()` 只能删除连续出现的重复元素。如果`fruit`包括

```
apple pear apple apple pear
```

简单地使用`fruit.unique()`将产生

```
apple pear apple pear
```

排序之后再做，就能给出

```
apple pear
```

如果希望删除某些确定的重复情况，我们可以提供一个谓词，刻画所需要删除的那种重复。例如，假定我们已经定义了一个二元谓词（18.4.2节）`initial2(x)`来比较`string`，如果串的起始字母是`x`就成立，对不以`x`开头的`string`返回`false`。给定

```
pear pear apple apple
```

通过

```
fruit.unique(initial2('p'));
```

这一调用，我们就能从`fruit`里删除以`'p'`开头的连续的重复串了，这将给出

```
pear apple apple
```

正如16.3.2节所特别指出的，我们有时会希望以反向顺序去看一个容器。对于`list`，还可以将所有元素反转过来，使第一个元素变成最后元素，等等，而且不复制元素。`reverse()`操作就是为此而提供的。给了

```
fruit:
    banana cherry lime strawberry
```

`fruit.reverse()`将产生出

```
fruit:
    strawberry lime cherry banana
```

从`list`里删除的元素都将被销毁。不过，应该注意，销毁一个指针并不意味着`delete`那个被指的对象。如果你希望一个指针的容器在将指针从容器删除时，或者在容器本身被销毁时，能够`delete`被这些指针所指的对象，你就必须自己写（17.8[13]）。

17.2.3 双端队列——`deque`

`deque`（其发音起伏就像`check`）就是一种双端的队列。也就是说，`deque`是一个优化了的序列，对其两端的操作效率类似于`list`，而其下标操作具有接近`vector`的效率：

```
template <class T, class A = allocator<T> > class std::deque {
    // 类型和操作与vector（16.3.3、16.3.5、16.3.6）一样，除了
    // capacity()和reserve()，但加上类似list的操作（17.2.2.2节）
};
```

“在中间”插入和删除元素具有与`vector`一样低效率，而不是类似`list`的效率。因此，`deque`一般用在那些需要由“两端”加入和删除的地方。例如，我们可能用一个`deque`模拟一段铁路或者游戏中的一叠扑克牌：

```
deque<car> siding_no_3;
deque<Card> bonus;
```


17.3 序列适配器

vector、*list*和*deque*序列不可能互为实现的基础而同时又不损失效率。但在另一方面，*stack*和*queue*则都可以在这三种基本序列的基础上幽雅而高效地实现。因此，*stack*和*queue*就没有定义为独立的容器，而是作为基本容器的适配器（adapter）。

容器适配器所提供的是原来容器的一个受限的界面。特别是适配器不提供迭代器，提供它们的意图就是为了只经由它们的专用界面使用。

从一个容器出发建立容器适配器的技术，对于为了用户需要，以非侵入的方式调整一个类的界面是非常有用的。

17.3.1 堆栈——*stack*

*stack*容器在 `<stack>` 中定义。它非常简单，以至描述它的最好方式就是给出一个实现：

```
template <class T, class C = deque<T> > class std::stack {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit stack(const C& a = C()) : c(a) {}

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```

可见，*stack*就是某容器的一个简单界面，容器的类型作为模板参数传递给*stack*。*stack*所做的全部事情就是从它的容器的界面中删除所有非*stack*操作，并为*back()*、*push_back()*和*pop_back()*重新确定人们所习惯的名字*top()*、*push()*和*pop()*。

按默认规定，*stack*用一个*deque*保存自己的元素，但也可以采用任何提供了*back()*、*push_back()*和*pop_back()*的序列。例如，

```
stack<char> s1;           // 用deque<char>保存char类型的元素
stack<int, vector<int>> s2; // 用vector<int>保存int类型的元素
```

也可以通过提供一个现存容器的方式对一个*stack*进行初始化。例如，

```
void print_backwards(vector<int>& v)
{
    stack<int, vector<int>> state(v); // 从v初始化状态
    while (state.size()) {
        cout << state.top();
        state.pop();
    }
}
```

不过，这个参数容器的所有元素都要复制，所以，提供一个现存容器可能代价很大。

加入`stack`的元素实际是通过`push_back()`放入那个用于存储元素的容器里的。因此,只要机器中还有可用内存供那个容器申请(使用它的分配器;19.4节),`stack`就不会上溢。

在另一方面,`stack`确实可能下溢:

```
void f()
{
    stack<int> s;
    s.push(2);
    if (s.empty()) {                // 下溢是可防护的
        // 不弹出
    }
    else {                          // 不是不可能的
        s.pop(); // 可以: s.size() 变成0
        s.pop(); // 无定义效果, 可能极糟
    }
}
```

注意,这里并不是用`pop()`弹出一个元素而后使用。相反,应该用`top()`访问,而后在不再需要时用`pop()`。这也不算不方便,还有,在不需要`pop()`时用这种方式更有效:

```
void f(stack<char>& s)
{
    if (s.top() == 'c') s.pop();    // 删除开头的 'c'
    // ...
}
```

与完整开发出的容器不同,`stack`(以及其他容器适配器)都没有分配器模板参数。相反,`stack`及其用户将依靠实现`stack`的容器所用的分配器。

17.3.2 队列——`queue`

`queue`在`<queue>`里定义,它也是一个容器的界面,该容器应该允许在`back()`处插入新元素,且能从`front()`提取出来:

```
template <class T, class C = deque<T> > class std::queue {
protected:
    C c;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit queue(const C& a = C()) : c(a) {}

    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }

    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }

    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }

    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

按照默认规定,`queue`用`deque`保存自己的元素,但是任何提供了`front()`、`back()`、

`push_back()` 和 `pop_front()` 的序列都可以用。因为 `vector` 没有提供 `pop_front()` 操作, 所以 `vector` 不能用做 `queue` 的基础容器。

`queue` 大概用在每个系统里的某些地方, 人们可以以如下方式为一个简单的消息传递系统定义一种服务:

```
struct Message {
    // ...
};

void server(queue<Message>& q)
{
    while(!q.empty()) {
        Message& m = q.front(); // 取出保存的消息
        m.service();             // 为满足请求去调用某个函数
        q.pop();                 // 销毁消息
    }
}
```

消息用 `push()` 放入 `queue` 里。

如果请求程序和服务程序在不同进程或者线程中运行, 对于队列的访问就必须提供某种形式的同步。例如,

```
void server2(queue<Message>& q, Lock& lck)
{
    while(!q.empty()) {
        Message m;
        {
            LockPtr h(lck);           // 只在提取消息时掌握着锁 (见14.4.1节)
            if (q.empty()) return;     // 其他程序取走了消息
            m = q.front();
            q.pop();
        }
        m.service();                  // 调用为请求提供服务的函数
    }
}
```

在C++里, 或一般说, 在世界的其他地方, 都还没有对并行性和锁的标准定义。请看一看你的系统已经提供了什么, 以及如何从C++里去访问它们 (17.8[8])。

17.3.3 优先队列——`priority_queue`

`priority_queue` 也是一种队列, 其中的每个元素被给定了一个优先级, 以此来控制元素达到 `top()` 的顺序:

```
template <class T, class C = vector<T>, class Cmp = less<typename C::value_type> >
class std::priority_queue {
protected:
    C c;
    Cmp cmp;
public:
    typedef typename C::value_type value_type;
    typedef typename C::size_type size_type;
    typedef C container_type;

    explicit priority_queue(const Cmp& a1 = Cmp(), const C& a2 = C())
        : c(a2), cmp(a1) { make_heap(c.begin(), c.end(), cmp); } // 见18.8节
```

```

template <class In>
priority_queue(In first, In last, const Cmp& = Cmp(), const C& = C());

bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }

const value_type& top() const { return c.front(); }

void push(const value_type&);
void pop();
};

```

`priority_queue`的声明可以在 `<queue>` 里找到。

按默认规定, `priority_queue`简单地用 `<` 运算符做元素比较, `top()` 返回最大的元素:

```

struct Message {
    int priority;
    bool operator<(const Message& x) const { return priority < x.priority; }
    // ...
};

void server(priority_queue<Message>& q, Lock& lck)
{
    while(!q.empty()) {
        Message m;
        { LockPtr h(lck);          // 只在提取消息时掌握着锁 (见14.4.1节)
          if (q.empty()) return;    // 其他程序取走了消息
          m = q.top();
          q.pop();
        }
        m.service(); // 调用为请求提供服务的函数
    }
}

```

这个例子与`queue`的例子(17.3.2节)不同,在这里,具有最高优先级的`messages`将首先得到服务。对具有同样优先级的元素,到达队列顶端的顺序没有明确定义,如果两个元素间互相间都在优先级上高于对方,那么就认为它们的优先级相同(17.4.1.5节)。

可以通过模板参数为元素比较 `<` 提供一个替代物。例如,我们可以按照不区分大小写的方式排列字符串,只要用`pq.push()`将它们放入优先队列

```
priority_queue<string, vector<string>, Nocase> pq; // 用Nocase做比较 (17.1.4.1节)
```

而后用`pq.top()`和`pq.pop()`提取它们。

给定不同模板参数,由模板定义的对象具有不同的类型(13.6.3.1节)。例如,

```
priority_queue<string>& pql = pq; // 错误: 类型不匹配
```

通过将一个具有适当类型的完成比较工作的对象作为构造函数的参数,就可以提供另一种比较准则,而且不会影响`priority_queue`的类型。例如,

```

struct String_cmp { // 用于在运行时表示比较准则的类型
    String_cmp(int n = 0); // 使用比较准则n
    // ...
};

typedef priority_queue<string, vector<string>, String_cmp> Pqueue;

void g(Pqueue& pq) // pq用String_cmp()做比较
{

```

```

Pqueue pq2(String_cmp(nocase));
pq = pq2; // 可以: pq和pq2类型相同, 现在pq也用String_cmp(nocase) 做比较
}

```

保持元素有序不会没有代价, 但这个代价并不大。实现`priority_queue`的一种有用方式是采用一个树结构保存元素的相对位置。这能给出 $O(\log(n))$ 代价的`push()`和`pop()`操作。

按照默认约定, `priority_queue`用一个`vector`保存它的元素, 但任何能提供`front()`、`push_back()`和`pop_back()`, 并能使用随机访问迭代器的序列都可以用。`priority_queue`最可能是用一个`heap` (18.8节) 实现。

17.4 关联容器

关联数组是用户定义类型中最常见的也最有用的一类。事实上, 在主要关注文字处理和符号处理的语言里, 关联数组甚至常是一种内部类型。关联数组也常被称为映射 (map), 有时被称为字典 (dictionary), 其中保存的是值的对偶。给定了一个称为关键码的值, 我们就能访问另一个称为映射值的值。可以将关联数组想像为一个下标不必是整数的数组:

```

template<class K, class V> class Assoc {
public:
    V& operator[] (const K&); // 返回对应于K的V的引用
    // ...
};

```

这样, 一个类型为`K`的关键码就起到作为类型为`V`的映射值的名字的作用。

关联容器是关联数组概念的推广。`map`就是传统的关联数组, 其中与每个关键码相关联的有唯一的~一个值。`multimap`是允许元素中出现重复关键码的关联数组, `set`和`multiset`也可以看做是退化的关联数组, 其中没有与关键码相关联的值。

17.4.1 映射——map

一个`map`就是一个 (关键码, 值) 对偶的序列, 它提供基于关键码的快速提取操作。每个关键码至多保持一个值, 换句话说, `map`中的关键码具有惟一性。`map`提供双向迭代器 (19.2.1节)。

`map`要求其关键码类型提供一个小于操作 (17.1.4.1节), 以保持自己元素的有序性。所以, 迭代通过`map`时就是按顺序的。对于那些没有明显顺序的元素, 或者不必保持容器有序的情况, 我们可以考虑用`hash_map` (17.6节)。

17.4.1.1 类型

`map`有普通容器都提供的成员类型 (16.3.1节), 还有几个与其特殊功能有关的类型:

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key,T>>>
class std::map {
public:
    // 类型:
    typedef Key key_type;
    typedef T mapped_type;
    typedef pair<const Key, T> value_type;
    typedef Cmp key_compare;

```

```

typedef A allocator_type;

typedef typename A::reference reference;
typedef typename A::const_reference const_reference;
typedef implementation_defined1 iterator;
typedef implementation_defined2 const_iterator;

typedef typename A::size_type size_type;
typedef typename A::difference_type difference_type;

typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
// ...
};

```

请注意, *map* 的 *value_type* 是 (关键码, 值) 的 *pair*, 映射值的类型用 *mapped_type* 表示。这样, 一个 *map* 也就是 *pair<const Key, mapped_type>* 的一个序列。

与平常一样, 迭代器的确切类型由实现定义。因为 *map* 最可能被实现为某种形式的树, 这些迭代器通常需要提供某些形式的树遍历。

反向迭代器通过标准的 *reverse_iterator* 模板创建 (19.2.5 节)。

17.4.1.2 迭代器和对偶

map 提供了一集普通的返回迭代器的函数 (16.3.2 节):

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T>>> class map {
public:
    // ...
    // 迭代器:

    iterator begin();
    const_iterator begin() const;

    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;

    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};

```

在 *map* 上的迭代器也就是在以 *pair<const Key, mapped_type>* 为元素的序列上的迭代器。举个例子, 我们可能用如下方式打印出一个电话簿的各项内容:

```

void f(map<string, number>& phone_book)
{
    typedef map<string, number>::const_iterator CI;
    for (CI p = phone_book.begin(); p != phone_book.end(); ++p)
        cout << p->first << '\t' << p->second << '\n';
}

```

一个 *map* 的迭代器按照 *map* 关键码的递增顺序给出它的元素 (17.4.1.5 节)。因此, 这个 *phone_book* 的项将按照字典顺序输出。

对任何 *pair*, 我们总用 *first* 和 *second* 索引其第一个和第二个元素, 无论其类型是什么:

```

template <class T1, class T2> struct std::pair {
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair() : first(T1()), second(T2()) { }
    pair(const T1& x, const T2& y) : first(x), second(y) { }
    template<class U, class V>
        pair(const pair<U, V>& p) : first(p.first), second(p.second) { }
};

```

提供最后一个构造函数是为了能做`pair`的转换（13.6.2节）。例如，

```

pair<int, double> f(char c, int i)
{
    pair<char, int> x(c, i);
    // ...
    return x; // 需要pair<char, int> 到pair<int, double> 的转换
}

```

在`map`里，关键码是其对偶的第一个元素，映射值是第二个。

`pair`的用途也不限于`map`的实现，它本身也是一个标准库类。`pair`的定义可以在 `<utility>` 里找到。这里还提供了—个用以方便地创建`pair`的函数：

```

template <class T1, class T2> pair<T1, T2> std::make_pair (T1& t1, T2& t2)
{
    return pair<T1, T2> (t1, t2);
}

```

在默认情况下，`pair`将被用其元素类型的默认值进行初始化。特别的，这也就意味着将内部类型的元素初始化为0（5.1.1节），将`string`初始化为空串（20.3.4节）。如果一个类型没有默认构造函数，要将其作为`pair`的元素时，就必须显式提供创建`pair`的参数。

17.4.1.3 下标

`map`的特征性操作就是采用下标运算符提供的关联查找：

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator< pair<const Key, T> > >
class map {
public:
    // ...
    mapped_type& operator[] (const key_type& k); // 用关键码k访问元素
    // ...
};

```

下标运算符将关键码作为下标去执行查找，并返回对应的值。如果不存在这个关键码，它就将一个具有该关键码和`mapped_type`类型默认值的元素插入这个`map`。例如，

```

void f()
{
    map<string, int> m; // map开始时为空
    int x = m["Henry"]; // 为Henry建立新项，初始化为0，返回0
    m["Harry"] = 7;    // 为Harry建立新项，初始化为0并赋值7
    int y = m["Henry"]; // 返回Henry对应项的值
    m["Harry"] = 9;    // 将Harry对应项的值修改为9
}

```

作为更实际一些的例子，考虑一个求和的计算程序，它对一些以（表格项名，值）对偶的形式输入的表格项求和，例如，

```
nail 100 hammer 2 saw 3 saw 4 hammer 7 nail 1000 nail 250
```

同时还求出每个表格项的和。主要工作都可以在将（表格项名称，值）对偶读入`map`的过程中完成：

```
void readitems (map<string, int>& m)
{
    string word;
    int val = 0;
    while (cin >> word >> val) m[word] += val;
}
```

下标运算`m[word]` 确定正确的（`string`, `int`）对偶，且返回对于其`int`部分的引用。这段代码还利用了新元素的`int`值自动设置为0的默认规定。

通过`readitems()` 创建起的`map`，可以在最后通过一个常规循环输出：

```
int main()
{
    map<string, int> tbl;
    readitems (tbl);

    int total = 0;
    typedef map<string, int>::const_iterator CI;
    for (CI p = tbl.begin(); p != tbl.end(); ++p) {
        total += p->second;
        cout << p->first << '\t' << p->second << '\n';
    }

    cout << "-----\ntotal\t" << total << '\n';

    return !cin;
}
```

对于上面给的输入，对应输出是：

```
hammer 9
nail 1350
saw 7
-----
total 1366
```

请注意，这里的项目将按照字典顺序打印出来（17.4.1节、17.4.1.5节）。

下标操作必须在`map`里找到对应的关键码，这样做的代价当然不可能像数组的整数下标那么低。代价是 $O(\log(\text{size_of_map}))$ ，对于大部分应用，这还是可以接受的。对于那些认为这种代价太高的应用，通常可以考虑采用散列容器（17.6节）。

在没有找到关键码时，`map`的下标操作将加进一个默认元素。因此，不能针对`const map`做`operator[]()`。进一步说，只有当`mapped_type`（值类型）有默认值时才能使用下标操作。如果程序员只是希望看一看某个关键码是否存在，那就可以用`find()`操作（17.4.1.6节）去找这个关键码，这样就不会修改`map`。

17.4.1.4 构造函数

`map`提供了常规性的一组完整的构造函数等（16.3.4节）：


```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T> > >
class map {
public:
    // ...
    // 构造/复制/析构等

    explicit map(const Cmp& = Cmp(), const A& = A());
    template <class In> map(In first, In last, const Cmp& = Cmp(), const A& = A());
    map(const map&);

    ~map();

    map& operator=(const map&);

    // ...
};

```

复制一个容器隐含着为它的所有元素分配空间，并完成每个元素的复制工作（16.3.4节）。这样做的代价可能非常高，只有迫不得已时才应该去做它。因此，像`map`这样的容器一般通过引用传递。

通过两个输入迭代器描述的是一个以 `pair<const Key, T>` 的序列为参数的成员模板构造函数。它将取自这个序列的所有元素用 `insert()`（17.4.1.7节）插入到`map`里。

17.4.1.5 比较

为能在`map`中找到对应于给定关键码的元素，`map`必须能够做关键码比较。还有，迭代器也会按照递增的方式遍历`map`，所以，在插入元素时通常也要做元素比较（以便将元素放到表示`map`的树结构中）。

按默认规定，关键码比较采用 `<`（小于），但是，可以通过模板参数或者构造函数的参数提供替代运算（17.3.3节）。现在需要比较的应该是关键码，而`map`的`value_type`是（关键码，值）对偶。因此提供了`value_comp()`，以便能对这种对偶使用关键码比较：

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T> > >
class map {
public:
    // ...

    typedef Cmp key_compare;

    class value_compare : public binary_function<value_type, value_type, bool> {
    friend class map;
    protected:
        Cmp cmp;
        value_compare(Cmp c) : cmp(c) {}
    public:
        bool operator()(const value_type& x, const value_type& y) const
        { return cmp(x.first, y.first); }
    };

    key_compare key_comp() const;
    value_compare value_comp() const;
    // ...
};

```

例如，

```

map<string, int> m1;
map<string, int, Nocase> m2; // 描述比较类型 (17.1.4.1节)
map<string, int, String_cmp> m3; // 描述比较类型 (17.1.4.1节)
map<string, int, String_cmp> m4(String_cmp(literary)); // 传递比较对象

```

有了`key_comp()`和`value_comp()`成员函数,我们就可能要求对一个`map`的关键码和值做各种比较。做这件事的一种常见做法是将另外某个容器或者算法的比较准则传递给`map`。例如,

```

void f(map<string, int>& m)
{
    map<string, int> mm; // 按默认方式用 < 做比较
    map<string, int> mmm(m.key_comp()); // 用m的方式比较
    // ...
}

```

参见17.1.4.1节中有关如何定义特定比较的实例,18.4节有关函数对象的一般性讨论。

17.4.1.6 映射操作

`map` (实际上,有关一般关联容器)的关键思想就是基于关键码取得信息。`map`为此提供了一些特别的操作:

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T>>>
class map {
public:
    // ...
    // map操作

    iterator find(const key_type& k); // 找关键码为k的元素
    const_iterator find(const key_type& k) const;

    size_type count(const key_type& k) const; // 关键码为k的元素个数

    iterator lower_bound(const key_type& k); // 找第一个关键码为k的元素
    const_iterator lower_bound(const key_type& k) const;
    iterator upper_bound(const key_type& k); // 找第一个关键码大于k的元素
    const_iterator upper_bound(const key_type& k) const;

    pair<iterator, iterator> equal_range(const key_type& k);
    pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // ...
};

```

`m.find(k)` 操作产生指向关键码为`k`的元素的迭代器。如果不存在这种元素,返回的迭代器将是`m.end()`。对于单一关键码的容器,如`map`或`set`,作为结果的迭代器将指向这个惟一的关键码为`k`的元素。对于关键码不惟一的容器,例如`multimap`或`multiset`,结果迭代器将指向关键码为`k`的第一个元素。例如,

```

void f(map<string, int>& m)
{
    map<string, int>::iterator p = m.find("Gold");
    if (p != m.end()) { // 如果找到 "Gold"
        // ...
    }
    else if (m.find("Silver") != m.end()) { // 寻找 "Silver"
        // ...
    }
    // ...
}

```

对于`multimap` (17.4.2节), 找到第一个匹配可能不如找到所有匹配那么有用。`m.lower_bound(k)` 和`m.upper_bound(k)` 给出`m`中关键码为`k`的子序列的开始和结束位置。通常, 一个序列的结束位置就是超过序列中最后元素的下一个位置。例如,

```
void f(multimap<string, int>& m)
{
    multimap<string, int>::iterator lb = m.lower_bound("Gold");
    multimap<string, int>::iterator ub = m.upper_bound("Gold");
    for (multimap<string, int>::iterator p = lb; p != ub; ++p) {
        // ...
    }
}
```

用两个独立函数找出上界和下界, 这样做既不幽雅, 也缺乏效率。为此提供的`range_equal()` 可以一下子算出这两个结果。例如,

```
void f(multimap<string, int>& m)
{
    typedef multimap<string, int>::iterator MI;
    pair<MI, MI> g = m.equal_range("Gold");
    for (MI p = g.first; p != g.second; ++p) {
        // ...
    }
}
```

如果`lower_bound(k)` 没有找到`k`, 它就返回关键码大于`k`的第一个元素的迭代器, 或者在根本没有这种关键码大于`k`的元素时返回`end()`。`m.upper_bound()` 和`range_equal()` 也采用这种方式报告工作失败。

17.4.1.7 表操作

将一个值放入关联容器的最方便方式就是使用下标操作直接赋值。例如,

```
phone_book["Order department"] = 8226339;
```

这将使在`phone_book`里有了一个Order department的项目, 无论以前是否将它放进去过。也可以通过`insert()` 直接插入项目, 或者通过`erase()` 删除其中的项目:

```
template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T>>>
class map {
public:
    // ...
    // 表操作:

    pair<iterator, bool> insert(const value_type& val); // 插入(关键码, 值) 对
    iterator insert(iterator pos, const value_type& val); // pos只是个提示
    template <class In> void insert(In first, In last); // 从序列中插入

    void erase(iterator pos); // 删除被指元素
    size_type erase(const key_type& k); // 删除关键码为k的元素
    void erase(iterator first, iterator last); // 删除一个区间
    void clear(); // 删除所有元素

    // ...
};
```

操作`m.insert(val)` 试图将一个 (Key, T) 对`val`加入`m`。因为`map`要保持关键码的惟一性, 所以,

只有在`m`中不存在具有这个关键码的元素时，插入才能真正实行。`m.insert(val)` 的返回值是 `pair<iterator, bool>`，如果`val`被实际插入，这个`bool`值就是`true`。返回值中的迭代器引用的是`m`中一个元素，它保存着`val`的关键码`val.first`。例如，

```
void f(map<string, int>& m)
{
    pair<string, int> p99("Paul", 99);

    pair<map<string, int>::iterator, bool> p = m.insert(p99);
    if (p.second) {
        // "Paul"被插入
    }
    else {
        // 已有"Paul"
    }
    map<string, int>::iterator i = p.first;    // 指向m["Paul"]
    // ...
}
```

通常我们并不关心一个关键码是新插入的，还是在`insert()`之前在`map`中就有的。当我们关心这件事时，多半是因为我们想在其他什么地方（在`map`之外）记录这个值已经放入了`map`的事实。`insert()`还有另外两个版本，它们都不通过返回值指明是否实际完成了插入。

在`insert(pos, val)`中描述了一个位置，这只是想给实现提供一个线索，要它从`pos`开始查找`val`的关键码。如果这个线索很好，结果可能是显著的性能改进。如果线索很坏，那么你不使用它更好，无论是从写法上还是效率上都是如此。例如，

```
void f(map<string, int>& m)
{
    m["Dilbert"] = 3;    // 简单，可能效率不高
    m.insert(m.begin(), make_pair(const string("Dogbert"), 99));    // 丑陋
}
```

事实上，`[]`不只是`insert()`的一种方便表述形式。`m[k]`的结果等价于`(*(m.insert(makepair(k, V()))).first)).second`，其中`V()`是映射类型的默认值。如果你理解了这个等价表示，你大概就完全理解关联容器了。

由于`[]`总需要用`V()`，如果一个`map`的值类型没有默认值，你就无法对它使用下标操作了，这是标准关联容器的一个很不幸的限制。因为无论如何，对默认值的需求并不是关联容器的本质特征（17.6.2节）。

你可以删除某个关键码所描述的全部元素。例如，

```
void f(map<string, int>& m)
{
    int count = m.erase("Ratbert");
    // ...
}
```

返回的整型值是被删除元素的个数。特别的，当不存在要删除的关键码为`"Ratbert"`的元素时，`count`将是0。对于`multimap`和`multiset`，这个值也可能大于1。换一种方式，也可以通过给定指向元素的迭代器的方式删除一个元素或者一个区间中的元素。例如：

```
void g(map<string, int>& m)
{
```

```

    m.erase(m.find("Catbert"));
    m.erase(m.find("Alice"), m.find("Wally"));
}

```

显然，如果你已经有了要删除元素的迭代器，直接删除这个元素肯定比先通过关键码查找，而后再删除要快得多。在`erase()`之后，这个迭代器就不能再用了，因为它所指的元素已经不在那里了。调用`m.erase(b, e)`，其中`e`是`m.end()`，也不会出问题（只要`b`引用的是`m`的某个元素或者`m.end()`）。另一方面，调用`m.erase(p)`，其中`p`是`m.end()`则是一个严重错误，而且可能破坏这个容器。

17.4.1.8 其他函数

最后，`map`还提供了一些处理元素个数的常用函数和专门的`swap()`：

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T>>>
class map {
public:
    // ...
    // 容量:

    size_type size() const;           // 元素个数
    size_type max_size() const;       // map的最大可能规模
    bool empty() const { return size() == 0; }

    void swap(map&);
};

```

通常，`size()`或`max_size()`的返回值是元素的个数。

此外，`map`还提供了`==`、`!=`、`<`、`>`、`<=`、`>=`和`swap()`，它们都作为非成员函数：

```

template <class Key, class T, class Cmp, class A>
bool operator==(const map<Key, T, Cmp, A>&, const map<Key, T, Cmp, A>&);

// !=、<、>、<=、>= 类似

template <class Key, class T, class Cmp, class A>
void swap(map<Key, T, Cmp, A>&, map<Key, T, Cmp, A>&);

```

为什么有人会想去比较两个`map`？当我们去比较两个`map`时，我们通常想知道的不仅是这两个`map`是否不同，也想知道（如果它们不同）到底有什么不同。在这些情况下，我们不会用`==`或`!=`。当然，通过为每个容器提供`==`、`!=`和`swap()`等，我们就有可能写出能应用于各种容器的算法。例如，这些函数使我们能用`sort()`对一个`map`的`vector`排序，或者得到一个`map`的`set`。

17.4.2 多重映射——`multimap`

`multimap`很像`map`，只是它允许重复的关键码：

```

template <class Key, class T, class Cmp = less<Key>,
          class A = allocator<pair<const Key, T>>>
class std::multimap {
public:
    // 与map类似，除了:

    iterator insert(const value_type&); // 返回iterator, 不是pair

    // 无下标操作符[]
};

```

例如（用17.1.4.1节的*Cstring_less*比较C风格字符串）：

```
void f(map<char*, int, Cstring_less>& m, multimap<char*, int, Cstring_less>& mm)
{
    m.insert(make_pair("x", 4));
    m.insert(make_pair("x", 5)); // 无影响：已有"x"的项（17.1.4.7节）
    // 现在m["x"] = 4

    mm.insert(make_pair("x", 4));
    mm.insert(make_pair("x", 5));
    // mm里现在在("x", 4)和("x", 5)
}
```

这些情况也隐含着*multimap*无法像*map*那样支持通过关键码值的下标操作。*equal_range()*、*lower_bound()*和*upper_bound()*操作（17.4.1.6节）是用一个关键码访问多重元素值的基本手段。

很自然，在那些对同一个关键码存在多个值的场合，用*multimap*比用*map*更合适。这种情况出现的比许多人初次遇到*multimap*时所认为的更频繁些。在某些方面，*multimap*甚至比*map*更清晰，也更优美。

因为一个人可能有多个电话号码，电话簿就是*multimap*的好例子。我可能用下面方式打印我的电话号码：

```
void print_numbers(const multimap<string, int>& phone_book)
{
    typedef multimap<string, int>::const_iterator I;
    pair<I, I> b = phone_book.equal_range("Stroustrup");
    for (I i = b.first; i != b.second; ++i) cout << i->second << '\n';
}
```

对于*multimap*，*insert()*操作总是真正的插入，因此，*multimap::insert()*将返回一个迭代器，而不是像*map*那样的`pair<iterator, bool>`。为了统一性，标准库原本也可以为*map*和*multimap*提供一般形式的*insert()*，即使对于*multimap*而言那个`bool`值是多余的。换一种设计方式，也可以提供一个简单的*insert()*，对于这两种情况都不返回`bool`值，而是为*map*的用户提供其他方式来确定某个关键码是不是原来所没有的。这是一个具体例子，在这里不同的界面设计思想发生了冲突。

17.4.3 集合——*set*

一个*set*也可以被看做是一个*map*（17.4.1节），其中的值是无关紧要的，所以我们只保留了关键码。这样做只引起了用户界面的少许变动：

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key>>
class std::set {
public:
    // 与map类似，除了：
    typedef Key value_type;           // 关键码就是值
    typedef Cmp value_compare;
    // 无下标操作符[]
};
```

把*value_type*定义为*Key*类型（*key_type*，17.4.1.1节）是一个妙招，这将使使用*map*和*set*的代

码在大部分情况下完全一样。

请注意, *set* 依靠的是比较操作 (默认为 `<`) 而不是相等 (`==`)。这隐含着元素的相等需要由不相等定义 (17.1.4.1 节), 而且迭代通过 *set* 有一种定义良好的顺序。

与 *map* 类似, *set* 也提供了 `==`、`!=`、`<`、`>`、`<=`、`>=` 和 *swap*()。

17.4.4 多重集合——*multiset*

multiset 是一种允许重复关键码的 *set*:

```
template <class Key, class Cmp = less<Key>, class A = allocator<Key>>
class std::multiset {
public:
    // 与set类似, 除了:
    iterator insert(const value_type&); // 返回iterator, 不是pair
};
```

equal_range()、*lower_bound*() 和 *upper_bound*() 操作 (17.4.1.6 节) 是访问关键码重复出现的基本手段。

17.5 拟容器

内部数组 (5.2 节)、*string* (第20章)、*valarray* (22.4 节) 和 *bitset* (17.5.3 节) 里也保存元素, 因此, 在许多情况下也可以将它们看做容器。当然, 它们中的每个都缺乏标准容器界面的这些或者那些侧面, 所以, 这些“拟容器”不能像开发完整的容器 (如 *vector* 或 *list*) 那样具有完全的互换性。

17.5.1 串——*string*

basic_string 提供下标操作、随机访问迭代器以及容器所能提供的几乎所有记法上的方便之处 (第20章)。然而, *basic_string* 不像容器那样支持广泛的元素类型选择。它还为作为字符串的使用做了优化, 其典型使用方式与容器有显著的差异。

17.5.2 值向量——*valarray*

valarray (22.4 节) 是为数值计算而优化了的向量。因此, *valarray* 并不企图成为一个具有一般性的容器。*valarray* 提供了许多有用的数值操作。但是, 在标准容器所提供的操作中 (17.1.1 节), 它只提供了 *size*() 和下标操作 (22.4.2 节)。到 *valarray* 中的元素的指针是一种随机访问迭代器 (19.2.1 节)。

17.5.3 位集合——*bitset*

系统的一些方面 (例如输入流的状态 (21.3.3 节) 等) 常用一组表明二进制条件的标志表示, 例如好或坏、真或假、开或关。C++ 可以通过整数上的按位运算, 有效地支持小的标志集合的概念。这些运算包括 `&` (与)、`|` (或)、`^` (异或)、`<<` (左移) 和 `>>` (右移)。类 *bitset*<*N*> 推广了这个概念, 并通过提供了在 *N* 个二进制位的集合 (下标由 0 到 *N*-1) 上的各种操作来提供更进一步的方便, 这里的 *N* 需要在编译时已知。对于无法放进 *long int* 的二进制位的集合, 用一个 *bitset* 比直接用一些整数方便得多。对于更小的集合, 这里就可能有一个效率

上的权衡问题。如果你希望给二进制位命名，而不是给它们编号，请考虑用`set`（17.4.3节）、枚举（4.8节）或者位域（C.8.1节）。

`bitset<N>` 是 N 个二进制位的数组。`bitset`与`vector<bool>`（16.3.11节）的不同之处在于它的大小是固定的；与`set`（17.4.3节）的不同点在于它通过下标索引其中的二进制位，而不是通过关键词关联；与`vector<bool>` 和`set`都不同的地方在于它提供了同时对许多二进制位进行处理的操作。

通过内部指针不可能直接对单个的位寻址（5.1节），因此，`bitset`提供了一种位引用类型。这实际上也是一种一般性的技术，用于对由于某些原因而使内部指针不能适用的对象寻址：

```
template<size_t N> class std::bitset {
public:
    class reference {          // 对单个位的引用；
        friend class bitset;
        reference();
    public:                    // b[i] 引用第i+1个位；
        ~reference();
        reference& operator=(bool x);          // 用于b[i] = x;
        reference& operator=(const reference&); // 用于b[i] = b[j];
        bool operator~() const;                // 返回~b[i];
        operator bool() const;                // 用于x = b[i];
        reference& flip();                     // b[i].flip();
    };
    // ...
};
```

`bitset`模板在名字空间`std`里定义，通过`<bitset>`给出。

由于历史的原因，`bitset`在风格上与其他标准库类有些差异。例如，如果一个下标（也称为位定位值）越界，它将抛出一个`out_of_range`异常。这里没有提供迭代器。位定位从右向左进行编号，与机器字里位编号的方式相同，所以`b[i]`的值就是`pow(2, i)`。这样，一个`bitset`也可以看做是一个 N 位的二进制数：

定位：	9	8	7	6	5	4	3	2	1	0
bitset<10>(989):	1	1	1	1	0	1	1	1	0	1

17.5.3.1 构造函数

`bitset`可以用默认值创建，或者从一个`unsigned long int`出发，或者从`string`出发创建：

```
template<size_t N> class bitset {
public:
    // ...
    // 构造函数：
    bitset();                // N个二进制位0
    bitset(unsigned long val); // 二进制位来自val

    template<class Ch, class Tr, class A>          // Tr是一个字符特征（20.2节）
    explicit bitset(const basic_string<Ch, Tr, A>& str, // 二进制位来自串str
        typename basic_string<Ch, Tr, A>::size_type pos = 0,
        typename basic_string<Ch, Tr, A>::size_type n = basic_string<Ch, Tr, A>::npos);

    // ...
};
```


二进制位默认为0。如果提供了一个`unsigned long int`，这个整数里的各个位将被用于初始化`bitset`里的各个位（如果有的话）。`basic_string`（第20章）参数也做同样的事情，只是用字符'0'给出二进制位0，用字符'1'给出二进制位1，其他字符都将引起`invalid_argument`异常的抛出。按默认约定，完整的串将用于初始化。但按照`basic_string`的构造函数的风格（20.3.4节），用户也可以提供所希望使用的从`pos`到串尾或者到`pos + n`的区间。例如，

```
void f()
{
    bitset<10> b1; // 全0

    bitset<16> b2 = 0xaaaa;           // 1010101010101010
    bitset<32> b3 = 0xaaaa;           // 00000000000000001010101010101010

    bitset<10> b4("1010101010");      // 1010101010
    bitset<10> b5("1011011101110", 4); // 0111011110
    bitset<10> b6("1011011101110", 2, 8); // 0011011101

    bitset<10> b7("n0g00d");           // 抛出invalid_argument
    bitset<10> b8 = "n0g00d";          // 错误：不存在char* 到bitset的转换
}
```

`bitset`设计的关键想法之一就是为能够放进一个机器字的`bitset`提供优化的实现，其界面也反应了这一考虑。

17.5.3.2 位处理操作

`bitset`提供了许多访问单个的位或者处理集合中所有的位的操作：

```
template<size_t N> class std::bitset {
public:
    // ...
    // bitset操作:

    reference operator[] (size_t pos);           // b[i]

    bitset& operator&= (const bitset& s);         // 与
    bitset& operator|= (const bitset& s);         // 或
    bitset& operator^= (const bitset& s);         // 异或

    bitset& operator<<= (size_t n);               // 逻辑左移（填充0）
    bitset& operator>>= (size_t n);               // 逻辑右移（填充0）

    bitset& set();                               // 将所有位都设置为1
    bitset& set(size_t pos, int val = 1);         // b[pos]=val

    bitset& reset();                             // 将所有位都设置为0
    bitset& reset(size_t pos);                   // b[pos]=0

    bitset& flip();                              // 改变每个位的值
    bitset& flip(size_t pos);                    // 改变b[pos] 的值

    bitset operator~() const { return bitset<N>(*this).flip(); } // 做出补集
    bitset operator<< (size_t n) const { return bitset<N>(*this)<<=n; } // 做出左移的集合
    bitset operator>> (size_t n) const { return bitset<N>(*this)>>=n; } // 做出右移的集合

    // ...
};
```

如果下标越界，下标运算符将抛出`out_of_range`异常。不存在不加检查的下标操作。

这一些操作中返回的`bitset&`就是`*this`。返回`bitset`（而不是`bitset&`）的操作将先做出`*this`的一个副本，随之将操作应用于这个副本，最后返回结果。特别的是，这里的`<<`和`>>`

都是移位操作，而不是I/O操作。*bitset*的输出操作也用 `<<`，其参数是一个*ostream*和一个*bitset* (17.5.3.3节)。

在移位时采用的是逻辑移位（而不是循环移位），也就是说，有一些位会“从一端掉出”，另一些位置将得到默认0。注意，因为*size_t*是无符号类型，所以不能移动负数位。当然，这也就意味着 `b << -1` 是用一个极大的整数做移位，将会使*bitset* *b*的所有各个位都变成0。你的编译器有可能对此提出警告。

17.5.3.3 其他操作

*bitset*还支持许多常用操作，如*size()*、*==*、I/O等：

```
template<size_t N> class bitset {
public:
    // ...

    unsigned long to_ulong() const;

    template <class Ch, class Tr, class A> basic_string<Ch, Tr, A> to_string() const;

    size_t count() const;           // 值为1的二进制位个数
    size_t size() const { return N; } // 位数

    bool operator==(const bitset& s) const;
    bool operator!=(const bitset& s) const;

    bool test(size_t pos) const;    // 如果b[pos]为1则true
    bool any() const;              // 如果任何位为1则true
    bool none() const;             // 如果没有位为1则true
};
```

操作*to_ulong()*和*to_string()*是构造函数的逆运算。为避免非显式的转换，采用这种命名操作比提供转换函数更好些。如果*bitset*的值中有意义的二进制位无法用一个*unsigned long*表示，*to_ulong()*将抛出*overflow_error*异常。

*to_string()*操作将产生一个所需要类型的字符串，其中保存着一个字符 '0' 和 '1' 的序列，*basic_string*是用于实现串的模板（第20章）。我们可以用*to_string()*写出一个*int*的二进制表示：

```
void binary(int i)
{
    bitset<8*sizeof(int)> b = i;    // 假定8位字节（见22.2节）
    cout << b.template to_string<char, char_traits<char>, allocator<char>>() << '\n';
}
```

不幸的是，调用一个显式限定的成员模板需要用很详尽且罕见的语法形式（C.13.6节）。

除了成员函数之外，*bitset*还提供了二元运算符`&`（与）、`|`（或）、`^`（异或），以及普通的I/O运算符：

```
template<size_t N> bitset<N> std::operator&(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator|(const bitset<N>&, const bitset<N>&);
template<size_t N> bitset<N> std::operator^(const bitset<N>&, const bitset<N>&);

template <class charT, class Tr, size_t N>
basic_istream<charT, Tr>& std::operator>>(basic_istream<charT, Tr>&, bitset<N>&);
template <class charT, class Tr, size_t N>
basic_ostream<charT, Tr>& std::operator<<(basic_ostream<charT, Tr>&, const bitset<N>&);
```

因此我们就可以直接写出*bitset*，而不必首先将它转换到*string*。例如，

```

void binary(int i)
{
    bitset<8*sizeof(int)> b = i;    // 假定8位字节 (见22.2节)
    cout << b << '\n';
}

```

这将从左向右打印出用1和0表示的各个位，最高的二进制位在最左端。

17.5.4 内部数组

内部数组提供了下标，且以常规指针形式提供了随机访问迭代器（2.7.2节）。不过，一个数组并不知道它自己的大小，所以用户必须自己保存有关其大小的信息。一般说，数组没有提供标准成员操作和类型。

完全可能为常规数组做出一种外观形式，以提供标准容器那样的记法规范，而又不改变其低级特性，这种做法有时也很有用：

```

template<class T, int max> struct c_array {
    typedef T value_type;

    typedef T* iterator;
    typedef const T* const_iterator;

    typedef T& reference;
    typedef const T& const_reference;

    T v[max];
    operator T* () { return v; }

    reference operator[] (size_t i) { return v[i]; }
    const_reference operator[] (size_t i) const { return v[i]; }

    iterator begin() { return v; }
    const_iterator begin() const { return v; }

    iterator end() { return v+max; }
    const_iterator end() const { return v+max; }

    size_t size() const { return max; }
};

```

`c_array`模板并不是标准库的一部分。将它放在这里只是作为一个简单的例子，用以说明可以怎样将一个“外来的”容器融入标准库的框架之中。也可以通过`begin()`、`end()`等对它使用标准算法（第18章）。它仍可以在堆栈上分配，不间接地使用动态存储。还有，也可以将它传递给期望一个指针的C风格函数。例如，

```

void f(int* p, int sz);    // C风格

void g()
{
    c_array<int, 10> a;
    f(a, a.size());        // C风格的使用
    c_array<int, 10>::iterator p = find(a.begin(), a.end(), 777); // C++/STL风格的使用
    // ...
}

```

17.6 定义新容器

标准容器提供了一个用户可以添加的框架。在这一节里，我将阐述如何以适当的方式加

入一个容器，使之在合理的地方都能与标准容器互换使用。这个实现刻意做得非常实际，但并不是最优化的。容器界面选择得非常接近`hash_map`的现存的、已经在广泛使用的高质量实现。将`hash_map`放在这里是为了研究一个一般性的问题。如果要作为实际产品使用，还是请用有技术支持的`hash_map`。

17.6.1 散列映射——`hash_map`

`map`是一个关联容器，几乎可以接受任何类型作为它的元素类型。它做到这些要依靠一个比较元素的小于运算（17.4.1.5节）。然而，如果我们对关键码的类型知道得更多一些，我们常常就能通过实现一个散列函数（`hash`函数），将容器实现为一个散列表，以减少查找元素所需要的时间。

一个散列函数就是一个能快速地将一个值映射到一个下标的函数，而且能保证很少有两个不同的值映射到同一个下标。简而言之，散列表就是将值存在它所对应的下标之处，除非在这里已经存储了另一个值，如果真有就将它存到“邻近处”。只要相等判断比较快，找到存储在其下标处的值就会很快，到“邻近处”去找它也不慢。因此，对于大型容器而言，`hash_map`能够提供比`map`快5至10倍的元素查找速度是很常见的，尤其是在查找速度特别重要的地方。另一方面，如果`hash_map`选择了病态的散列函数，它也可能比`map`慢得多。

存在着许多实现散列表的方法。`hash_map`在界面设计上与标准关联容器有一些不同之处，但只是在那些为通过散列取得高性能而需要的地方。`map`和`hash_map`之间最本质的差异就在于：`map`对其元素类型要求有一个`<`，而`hash_map`要求一个`==`和一个散列函数。这样，`hash_map`的非默认创建方式必然就与`map`不同。例如，

```
map<string, int> m1;           // 用 < 比较串
map<string, int, Nocase> m2;    // 用Nocase() 比较串 (17.1.4.1节)

hash_map<string, int> hm1;      // 用Hash<string>() 散列 (17.6.2.3), 用 == 比较
hash_map<string, int, hfct> hm2; // 用hfct() 散列, 用 == 比较
hash_map<string, int, hfct, eql> hm3; // 用hfct() 散列, 用eql比较
```

采用散列查找的容器用一个或者几个表实现。除了保存它的元素外，容器还需要维持与每个散列值（下标，按照前面的解释）相关联的那一组值的情况，这些都通过一个散列表完成。大部分散列表在它的表“太满”（如75%满）时性能就会恶化。因此，下面要定义的`hash_map`在它接近太满时就会自动地改变自己的大小。当然，这种改变的成本可能很高，因此，能给定一个初始大小很有价值。

对于`hash_map`的第一个近似大致具有下面的样子：

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
class hash_map {
    // 与map类似，除了：

    typedef H Hasher;
    typedef EQ key_equal;

    hash_map(const T& dv = T(), size_type n = 101, const H& hf = H(), const EQ& = EQ());
    template<class In> hash_map(In first, In last,
        const T& dv = T(), size_type n = 101, const H& hf = H(), const EQ& = EQ());
};
```

简而言之，这就是`map`的界面（17.4.1.4节），只不过用`==` 和一个散列函数取代了`<`。

本书中，到此为止所有使用`map`的例子（3.7.4节、6.1节、17.4.1节）都可以转为使用`hash_map`，要做的就是将名字`map`换成`hash_map`。在许多情况下，从`map`改到`hash_map`可以很容易地通过一个`typedef`完成。例如：

```
typedef hash_map<string, record> Map;
Map dictionary;
```

要想进一步把`dictionary`的实际类型对其用户隐蔽起来，`typedef`也是很有用的。

虽然不具有严格的正确性，我觉得在`map`与`hash_map`之间的权衡也就是有关时间和空间的权衡。如果效率不成问题，根本就没有必要在它们中选择，两个都很好。对于大而密集使用的表，`hash_map`必然具有速度优势，只要空间不紧张就应该用它。即使空间紧张，我也可能会在选用“普通的”`map`之前，想想有没有别的办法节约空间。但还是应该经过实测，以避免挖空心思去优化原本选错了的代码是很重要的。

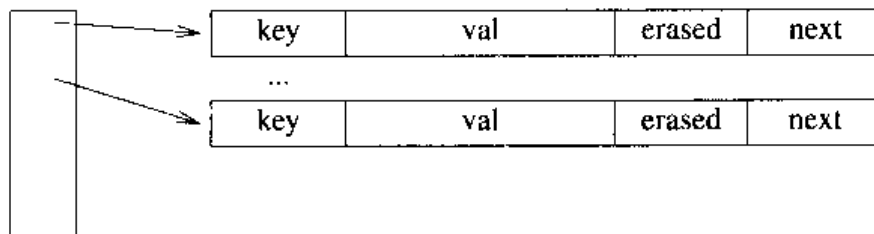
有效散列的关键在于散列函数的质量。如果无法找到一个好的散列函数，`map`就很容易在性能上超过`hash_map`。基于C风格字符串、`string`或整数的散列通常都非常有效。当然，还必须牢记，散列函数的有效性将依赖于被实际散列的那些值，这一点也具有决定性（17.8[35]）。在没有`<`定义，或者小于关系对打算使用的关键码不合适的地方，就必须使用`hash_map`。反过来说，散列函数不能像`<`那样定义，一个排序关系，如果保持元素的排列顺序很重要，那么就必须用`map`了。

与`map`一样，`hash_map`也提供了一个`find()`，使程序员可以确定某个关键码是否已经插入在表中。

17.6.2 表示和构造

`hash_map`有多种可取的互不相同的实现方式。这里，我使用一种具有合理的速度，而且各种重要操作都比较容易实现的方式。关键操作是构造函数、查找（运算符`[]`）、更改规模的操作以及删除元素的操作（`erase()`）。

这里所选择的简单实现方式依赖于一个散列表，它是一个指向表项（`entry`）的指针的`vector`。每个`Entry`中保存着一个`key`、一个`value`和一个指向下一个具有同样散列值的`Entry`的指针。`Entry`中还有一个`erased`位：



用声明来描述，它大致是这样：

```
template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
class hash_map {
    // ...
private:
    // 表示
    struct Entry {
```

```

        key_type key;
        mapped_type val;
        bool erased;
        Entry* next;    // 散列溢出链
        Entry(key_type k, mapped_type v, Entry* n)
            : key(k), val(v), erased(false), next(n) {}
    };

    vector<Entry> v;    // 实际表项
    vector<Entry*> b;    // 散列表: 到v里的指针

    // ...
};

```

请注意`erased`位。这里处理具有相同散列值的多个元素所采用的方式使元素的删除难以进行。所以，在调用`erase()`时并不是实际地将元素删除，我只是将那个元素标记为`erased`并忽略它，直到下一次更改表的规模时再去处理。

除了主要的数据结构之外，`hash_map`还需要一点为管理而使用的数据。当然，每个构造函数都必须去设置所有的这种数据。例如：

```

template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
class hash_map {
    // ...

    hash_map(const T& dv=T(), size_type n=101, const H& h=H(), const EQ& e=EQ())
        : default_value(dv), b(n), no_of_erased(0), hash(h), eq(e)
    {
        set_load();                // 默认
        v.reserve(max_load*b.size()); // 为增长保留空间
    }

    void set_load(float m = 0.7, float g = 1.6) { max_load = m; grow = g; }

    // ...
private:
    float max_load;                // 保持v.size() <= b.size() * max_load
    float grow;                    // 在必要时resize(bucket_count() * grow)
    size_type no_of_erased;        // 在v里由erased元素所占的项的个数
    Hasher hash;                   // 散列函数
    key_equal eq;                  // 相等

    const T default_value;        // 被[]所用的默认值
};

```

标准关联容器要求被映射类型有一个默认值（17.4.1.7节）。这个限制在逻辑上并不必要，也可能带来不方便。将默认值作为参数，使我们可以写：

```

hash_map<string, Number> phone_book1;           // 默认值: Number()
hash_map<string, Number> phone_book2(Number(411)); // 默认值: Number(411)

```

17.6.2.1 查找

最后，我们可以提供最关键的查找操作：

```

template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
class hash_map {
    // ...

```

```

mapped_type& operator[] (const key_type&);
iterator find(const key_type&);
const_iterator find(const key_type&) const;
// ...
};

```

为找到某个值，`operator[]()` 用散列函数为 `key` 找出对应这个散列表的下标，而后它检索一些项，直到找到与之匹配的 `key`。这个 `Entry` 里的 `value` 就是我们要找的。如果找不到这样的项，就插入一个默认值：

```

template<class Key, class T, class H = Hash<Key>,
         class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
hash_map<Key, T, H, EQ, A>::mapped_type&
hash_map<Key, T, H, EQ, A>::operator[] (const key_type& k)
{
    size_type i = hash(k) % b.size();           // 散列
    for (Entry* p = b[i]; p; p = p->next) // 在散列到 i 的项中搜索
        if (eq(k, p->key)) {                   // 找到
            if (p->erased) {                   // 重新插入
                p->erased = false;
                no_of_erased--;
                return p->val = default_value;
            }
            return p->val;
        }
    // 未找到:
    if (size_type(b.size() * max_load) <= v.size()) { // 如果“太满”
        resize(b.size() * grow);                // 增长
        return operator[] (k);                  // 重新散列
    }
    v.push_back(Entry(k, default_value, b[i])); // 加入 Entry
    b[i] = &v.back();                          // 指向新元素
    return b[i]->val;
}

```

与 `map` 不同，`hash_map` 并不依赖于从小于操作合成出的一个相等检测（17.1.4.1节），这是因为，在检查具有相同关键码值的循环中，需要反复调用 `eq()`。对于查找的性能而言这个循环至关重要，而对一些常见的明显关键码类型（如 `string` 或者 C 风格的字符串），额外比较的开销也非常可观。

当然也可以用一个 `set<Entry>` 来表示具有同样关键码的值集合。但是，如果我们有很好的散列函数（`hash()`）和一个规模适宜的表（`b`），大部分这种集合里都将只有一个元素。正因为这种情况，我才把这个集合的元素用 `Entry` 的 `next` 域链接起来（17.8[27]）。

注意，`b` 里保存着到 `v` 中元素的指针，元素被加入到 `v` 里面。一般说，`push_back()` 也可能引起重新分配，这样就使指向元素的指针变成非法（16.3.5节）。但是，在日前这种情况下，构造函数（17.6.2节）和 `resize()` 都小心地 `reserve()` 了足够的空间，因此不会发生未预见到的重新分配。

17.6.2.2 删除和更改规模

当表太满时，散列查找的效率会变得很低。为减少这种情况的出现，这里的散列表将由

下标运算符自动调用`resize()`扩张。`set_load()` (17.6.2节)提供了一种对发生规模改变的时机进行控制的方式。这里还提供了另一些函数,使程序员可以查看`hash_map`的状态:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
class hash_map {
    // ...

    void resize(size_type n);           // 将散列表的大小改为n
    void erase(iterator position);      // 删除被指元素

    size_type size() const { return v.size() - no_of_erased; } // 元素个数
    size_type bucket_count() const { return b.size(); }        // 散列表的规模
    Hasher hash_fun() const { return hash; }                   // 所用散列函数
    key_equal key_eq() const { return eq; }                     // 所用相等比较
    // ...
};
```

`resize()`操作是最关键的,它相当简单,但潜在的代价很高:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
void hash_map<Key, T, H, EQ, A>::resize(size_type s)
{
    size_type i = v.size();
    while (no_of_erased) {           // 实际删除erased元素
        if (v[--i].erased) {
            v.erase(&v[i]);
            --no_of_erased;
        }
    }

    if (s <= b.size()) return;
    b.resize(s);                     // 增加s ~ b.size()个指针
    fill(b.begin(), b.end(), 0);     // 全部项清0 (18.6.6节)
    v.reserve(s * max_load);         // 如果v需要重新分配,现在就做

    for (size_type i = 0; i < v.size(); i++) {           // 重新散列
        size_type ii = hash(v[i].key) % b.size();        // 散列
        v[i].next = b[ii];                               // 链接
        b[ii] = &v[i];
    }
}
```

如果需要的话,用户也可以“手工地”调用`resize()`,以保证这个代价发生在可预见的时刻。我发现`resize()`对有些应用非常重要,但是它并不是散列表的基本概念。有些实现策略根本就不需要它。

由于所有实际工作都在其他地方做了(而且只在散列表改变规模时),`erase()`变得极简单:

```
template<class Key, class T, class H = Hash<Key>,
        class EQ = equal_to<Key>, class A = allocator<pair<const Key, T>>>
void hash_map<Key, T, H, EQ, A>::erase(iterator p) // 删除被指向的元素
{
    if (p->erased == false) no_of_erased++;
    p->erased = true;
}
```


17.6.2.3 散列

为了完成这个`hash_map::operator[]()`，我们还需要定义`hash()`和`eq()`。散列函数最好是定义成一个函数对象的`operator()()`，其原因到18.6节才能更清楚：

```
template <class T> struct Hash : unary_function<T, size_t> {
    size_t operator()(const T& key) const;
};
```

一个好的散列函数以一个关键码为参数，返回一个整数，并使不同关键码有很大的可能性产生出不同的整数。选出一个好的散列函数是一种艺术。不过，将关键码的表示通过异或运算放入一个整数常常是一种可接受的方式：

```
template <class T> size_t Hash<T>::operator()(const T& key) const
{
    size_t res = 0;
    size_t len = sizeof(T);
    const char* p = reinterpret_cast<const char*>(&key); // 将对象作为字节序列去访问
    while (len--) res = (res<<1)^*p++; // 使用表示key的那些字节
    return res;
}
```

对`reinterpret_cast`（6.2.7节）的应用表明这里正在做某些令人厌烦的事情。如果我们对要散列的东西了解得更多，我们就能做得更好一些。特别的，如果在对象里包含指针，如果对象很大，或者如果成员的对齐要求使得表示中留下了无用空间（“空洞”），在这些情况下我们都可以做得更好一些^①（见17.8[29]）。

C风格的字符串就是一个指针（字符指针），而`string`里包含着指针。因此，必须要做针对它们的专门化：

```
typedef char* Pchar;

template<> size_t Hash<Pchar>::operator()(const Pchar& key) const
{
    size_t res = 0;
    Pchar p = key;
    while (*p) res = (res<<1)^*p++; // 用字符的int值
    return res;
}

template <class C>
size_t Hash<basic_string<C>>::operator()(const basic_string<C>& key) const
{
    size_t res = 0;

    typedef typename basic_string<C>::const_iterator CI;
    CI p = key.begin();
    CI end = key.end();

    while (p!=end) res = (res<<1)^*p++; // 用字符的int值
    return res;
}
```

① 这些情况也正是上面通用处理方式中存在重大缺陷之处。因此，对于一些最常见的情况，需要下面的特殊处理。作者没做的请读者自己考虑，例如对象里的对齐空洞会引起什么问题。——译者注

一个`hash_map`的实现将至少包含针对整数关键码和字符串关键码的散列函数。对那些更具冒险性的关键码类型，用户或许也必须予以帮助，提供合适的专门化散列函数。经验告诉我们，在选择散列函数时，做一些好的实测是必不可少的，直觉在这个领域里常常不那么管用。

要完成这个`hash_map`，我们还需要定义迭代器和不多的简单函数，这些都留做练习（17.8[34]）。

17.6.3 其他散列关联容器

为了一致性和完整性，`hash_map`还应该有与之匹配的`hash_set`、`hash_multimap`和`hash_multiset`。根据`hash_map`和`map`、`multimap`、`set`、`multiset`的定义，这些容器的定义也是很明显的，所以我也把它们留做练习（17.8[34]）。这些散列容器都有很好的公共领域版本和商用版本。对于实际程序设计，应该更倾向于用这些版本，而不是某个自己编排的版本，如我的这个。

17.7 忠告

- [1] 如果需要用容器，首先考虑用`vector`；17.1节。
- [2] 了解你经常使用的每个操作的代价（复杂性，大O度量）；17.1.2节。
- [3] 容器的界面、实现和表示是不同的概念，不要混淆；17.1.3节。
- [4] 你可以依据多种不同准则去排序和搜索；17.1.4.1节。
- [5] 不要用C风格的字符串作为关键码，除非你提供了一种适当的比较准则；17.1.4.1节。
- [6] 你可以定义这样的比较准则，使等价的但是不相同的关键码值映射到同一个关键码；17.1.4.1节。
- [7] 在插入和删除元素时，最好是使用序列末端的操作（`back`操作）；17.1.4.1节。
- [8] 当你需要在容器的前端或中间做许多插入和删除时，请用`list`；17.2.2节。
- [9] 当你主要通过关键码访问元素时，请用`map`或`multimap`；17.4.1节。
- [10] 尽量用最小的操作集合，以取得最大的灵活性；17.1.1节。
- [11] 如果要保持元素的顺序性，选用`map`而不是`hash_map`；17.6.1节。
- [12] 如果查找速度极其重要，选`hash_map`而不是`map`；17.6.1节。
- [13] 如果无法对元素定义小于操作时，选`hash_map`而不是`map`；17.6.1节。
- [14] 当你需要检查某个关键码是否在关联容器里的时候，用`find()`；17.4.1.6节。
- [15] 用`equal_range()`在关联容器里找出所有具有给定关键码的所有元素；17.4.1.6节。
- [16] 当具有同样关键码的多个值需要保持顺序时，用`multimap`；17.4.2节。
- [17] 当关键码本身就是你需要保存的值时，用`set`或`multiset`；17.4.3节。

17.8 练习

对本章中的一些练习，通过查看标准库实现的源文件可以找到解决方法。为你自己着想：在查看你所用的库的实现者如何解决这些问题之前，先试着自己去找出一种解决方法，而后再去看看你所用实现版本中的容器及其操作。

1. (*2.5) 理解 $O()$ 记法（17.1.2节）。对于标准容器的各种操作做一些测量，确定它们所涉及的常量因子。

2. (*2) 许多电话号码无法放入`long`里。写一个`phone_number`类型和一个`phone_number`的容器类，它提供了--组有用的操作。
3. (*2) 写一个程序，它能按照字典顺序列出一个文件中的所有不同单词。做两个版本：一个将简单的空白分隔的字符序列看做单词，另一个将任意非字母序列分隔的字母序列看做单词。
4. (*2.5) 实现一个简单的单人纸牌游戏。
5. (*1.5) 实现--个简单的检查单词是否为回文的测试程序（回文就是其表示为对称的字符串，例如，*ada*、*otto*、*tut*等）。实现一个简单的检查整数是否回文的测试程序。实现一个检查一个句子是否回文的程序。推广它们。
6. (*1.5)（只）用两个`stack`定义一个队列。
7. (*1.5) 定义一个类似`stack`（17.3.1节）的堆栈，它不复制它的基础容器，但是允许在它的元素上进行迭代。
8. (*3) 你的计算机应该能通过线程、作业或者进程的概念支持并发性活动。弄清楚这些是如何完成的。这些并行机制必然有一个概念用于锁定，以防止两个作业同时访问同一处存储。利用机器的锁定机制实现一个锁类。
9. (*2.5) 从输入中读入日期的序列，例如*Dec85*、*Dec50*、*Jan76*等，而后输出它们，使后面的日期先输出。日期的格式是三个字符的月份名加上两个数字的年份。假定所有日期都来自同一个世纪。
10. (*2.5) 推广日期的输入格式，允许日期是*Dec1985*、*12/3/1990*、*(Dec, 30, 1950)*、*3/6/2001*这样的格式。修改17.8[9] 处理这些新格式。
11. (*1.5) 利用`bitset`输出某些数值的二进制表示，包括0、1、-1、18、-18及最大的正`int`。
12. (*1.5) 利用`bitset`表示某一天一个班级里的哪些学生到场。读入一系列12天的`bitset`，确定哪些学生每天都来，哪些学生至少来过8天。
13. (*1.5) 写一个指针的`List`，当它自身被销毁时，或者当元素从该`List`中删除时，它总能`delete`被指的对象。
14. (*1.5) 给定`stack`对象，按顺序打印出它的元素，而且不改变这个堆栈的值。
15. (*2.5) 完成`hash_map`（17.6.1节）。这涉及到实现`find()`和`equal_range()`，再设计一种方式测试这个整个的模板。至少用一种默认散列函数不适应的类型测试`hash_map`。
16. (*2.5) 按照标准`list`的风格实现和测试一个表。
17. (*2) 有时`list`的空间开销可能成为问题。按照标准容器的风格写一个单链表并测试它。
18. (*2.5) 实现一个类似标准`list`的表，但它支持下标操作。对各种各样的表，比较其下标操作的代价与用同样长度的`vector`上下标操作的代价。
19. (*2) 实现归并两个容器的模板函数。
20. (*1.5) 确定所给C风格字符串是否回文。确定在字符串的开始是否至少有三个单词形成回文。
21. (*2) 读入一系列的 (*name*, *value*) 对，产生一个排好序的 (*name*, *total*, *mean*, *median*) 四元组的表，打印这个表。
22. (*2.5) 确定你所用的实现中各种标准容器的空间开销。
23. (*3.5) 考虑什么是使用最少空间的`hash_map`的一种合理实现策略。考虑一种具有最小查找时间的`hash_map`的合理实现策略。在各种情况中，考虑你为接近理想状态而可能想除掉

的操作（分别从没有空间额外开销或者没有查找的额外开销考虑）。提示：有关散列表的文献很多。

24. (*2) 设计一种处理`hash_map`溢出的策略（也就是说，不同关键码值散列到同一散列值），使得`equal_range`很容易实现。
25. (*2.5) 估计`hash_map`的空间开销，然后做实测，对估计值与实测值做一个比较。比较你所做的`hash_map`和你所用的实现中的`map`的空间开销。
26. (*2.5) 做出你自己的`hash_map`运行剖面（profile），看时间都花在什么地方。对你所用实现中的`map`和广泛流行的`hash_map`做这件事。
27. (*2.5) 基于`vector<map<K, V>*>`实现一个`hash_map`，使每个`map`里保存着所有具有相同散列值的关键码。
28. (*3) 基于Splay树（参见D. Sleator and R. E. Tarjar: *Self-Adjusting Binary Search Trees*, JACM, Vol. 32, 1985）实现一个`hash_map`。
29. (*2) 给定下面所描述的与串类似的数据结构：

```
struct St {
    int size;
    char type_indicator;
    char* buf;           // 指向size个字符
    St(const char* p);    // 分配和填充buf
};
```

建立1000个`St`并用它们作为`hash_map`的关键码。设计一个程序去实测`hash_map`的性能。特别为`St`关键码写一个散列函数（`Hash`，17.6.2.3节）。

30. (*2) 给出至少四种不同方式，实现从`hash_map`中删除`erased`元素的工作。你可以用标准库算法（3.8节，第18章）来避免显式的循环。
31. (*3) 实现一个立即删除元素的`hash_map`。
32. (*2) 在17.6.2.3节给出的散列函数并不总能考虑到关键码表示的全部内容。什么时候表示中的一些部分会被忽略？写出一个散列函数，使它总能考虑关键码表示的全部内容。给出一个例子，说明在什么时候忽略关键码中某个部分可能是明智的。写一个散列函数，使它只基于关键码中被认为有关的部分去计算散列值。
33. (*2.5) 散列函数的代码大致都差不多：用一个循环取得更多数据并散列之。定义一个`Hash`（17.6.2.3节），使它能通过反复调用一个函数的方式取得数据，使用户可以按照每种类型定义这个函数。例如：

```
size_t res = 0;
while (size_t v = hash(key) + res = (res << 3) ^ v;
```

在这里，用户可以为需要散列的每个类型`K`定义`hash(K)`。

34. (*3) 有了`hash_map`的某个实现，请实现`hash_multimap`、`hash_set`和`hash_multiset`。
35. (*2.5) 写一个散列函数，其意图是将均匀分布的`int`值映射到某个大小约为1024的散列表的散列值。有了这个函数后，设计一组1024个关键码值，使它们都映射到同一个值。

第18章 算法和函数对象

形式正在开放。
——工程师的俗语

引言——标准算法综述——序列——函数对象——谓词——算术对象——约束器——成员函数对象——*for_each*——查找元素——*count*——序列比较——检索——复制——*transform*——取代或删除元素——序列填充——重排——*swap*——序列排序——*binary_search*——*merge*——集合运算——*min*和*max*——堆——排列——C风格算法——忠告——练习

18.1 引言

容器本身并不那么让人感兴趣。为了真能有用，容器还必须提供一些基本操作，例如，确定其大小、复制、排序、查找元素等。幸运的是，标准库提供了许多算法，它们能服务于容器用户的最普遍、最基本的需要。

本章将综述这些标准算法，并给出一些使用它们的例子，展示用于在C++语言里描述算法的基本原则和技术，并解释若干关键算法的细节。

函数对象提供了一种机制，使用户可以通过它去定制标准算法的行为。函数对象能为算法提供对用户数据进行操作所需要的关键性信息。因此，这里也要强调如何定义和使用函数对象。

18.2 标准库算法综述

乍看起来，标准库算法似乎有些使人不知所措。其实它们也不过就是60个，我看见过许多的类有比这还多的成员函数。进一步说，许多算法具有共同的基本行为方式以及共同的界面风格，这些都使它们更容易理解。就像语言特征一样，程序员应该去使用那些自己实际需要并已经理解了算法，而且只使用它们。在一个程序里使用最大数目的标准算法绝不能得到任何回报，以最聪明或最隐晦的方式去使用标准算法也不会有任何回报。切记，写代码的一个基本目的就是使它意义清晰，以便后来者阅读，而这个后来者也可能就是几年之后的你自己。在另一方面，在对容器中的元素做什么事情时，也应该想一想，这个动作是否能以标准库风格的算法去表述。这种算法可能已经存在。如果你不考虑采用通用算法的方式的话，你将会再一次来发明这个轮子。

每个算法都表述为一个模板函数（13.3节）或者一组模板函数。按照这种方式，一个算法能在许多不同种类的序列上操作，这些序列中包含着各种类型的元素。返回迭代器（19.1节）的算法都用其输入序列的结束作为失败指示。例如，

```

void f(list<string>& ls)
{
    list<string>::const_iterator p = find(ls.begin(), ls.end(), "Fred");
    if (p == ls.end()) {
        // 未发现 "Fred"
    }
    else {
        // 这里p指向 "Fred"
    }
}

```

这些算法都不对其输入或者输出做范围检查。应该通过其他途径防止越界错误(18.3节、19.3节)。当算法返回一个迭代器时,这个迭代器总具有和算法的某个输入一样的类型。特别地,算法的参数控制着该算法是返回`const_iterator`还是非`const iterator`。例如,

```

void f(list<int>& li, const list<string>& ls)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);
    list<string>::const_iterator q = find(ls.begin(), ls.end(), "Ring");
}

```

标准库里的算法覆盖了在容器上的各种最具普遍性的操作,例如遍历、排序、检索、插入或者删除元素等。标准算法都在名字空间`std`里,它们的声明可以在`<algorithm>`里看到。有趣的是,大部分最常用的算法都极其简单,以至使这些模板函数都能在线处理。这也意味着,由这些算法所表述的循环能够从积极的函数优化中获益。

标准函数对象也在名字空间`std`里,但它们的声明在`<functional>`里。函数对象的设计也特别考虑了在线替换问题。

非修改性序列操作用于从序列中获取信息或者找出某些元素在序列中的位置:

非修改性的序列操作 (18.5节) <code><algorithm></code>	
<code>for_each()</code>	对序列中每个元素执行某个操作
<code>find()</code>	在序列中找出某个值的第一个出现
<code>find_if()</code>	在序列中找出符合某谓词的第一个元素
<code>find_first_of()</code>	在一序列中找出另一个序列里的值
<code>adjacent_find()</code>	找出相邻的一对值
<code>count()</code>	在序列中统计某个值出现的次数
<code>count_if()</code>	在序列中统计与某谓词匹配的次数
<code>mismatch()</code>	找出使两个序列相异的第一个元素
<code>equal()</code>	如果两个序列对应元素都相同则真
<code>search()</code>	找出一序列作为子序列的第一个出现位置
<code>find_end()</code>	找出一序列作为子序列的最后一个出现位置
<code>search_n()</code>	找出一序列作为子序列的第 n 个出现位置

许多算法允许用户描述对每一个或者每一对元素执行的实际动作,这就使这些算法比它们初看起来更有用得多了。特别地,用户可以为相同或者相异提供判断准则(18.4.2节)。在任何合理之处,这些函数都以最常见最有用的操作作为默认值。

除了它们都可能修改序列中元素的值之外,修改性的序列操作之间的共性很少:

修改性的序列操作 (18.6节) <algorithm>	
<i>transform()</i>	将操作应用于序列中的每个元素
<i>copy()</i>	从序列的第一个元素起进行复制
<i>copy_backward()</i>	从序列的最后元素起进行复制
<i>swap()</i>	交换两个元素
<i>iter_swap()</i>	交换由迭代器所指的两个元素
<i>swap_ranges()</i>	交换两个序列中的元素
<i>replace()</i>	用一个给定值替换一些元素
<i>replace_if()</i>	替换满足谓词的一些元素
<i>replace_copy()</i>	复制序列时用一个给定值替换元素
<i>replace_copy_if()</i>	复制序列时替换满足谓词的元素
<i>fill()</i>	用一个给定值取代所有元素
<i>fill_n()</i>	用一个给定值取代前 n 个元素
<i>generate()</i>	用一个操作的结果取代所有元素
<i>generate_n()</i>	用一个操作的结果取代前 n 个元素
<i>remove()</i>	删除具有给定值的元素
<i>remove_if()</i>	删除满足一个谓词的元素
<i>remove_copy()</i>	复制序列时删除给定值的元素
<i>remove_copy_if()</i>	复制序列时删除满足谓词的元素
<i>unique()</i>	删除相邻的重复元素
<i>unique_copy()</i>	复制序列时删除相邻的重复元素
<i>reverse()</i>	反转元素的次序
<i>reverse_copy()</i>	复制序列时反转元素的次序
<i>rotate()</i>	循环移动元素
<i>rotate_copy()</i>	复制序列时循环移动元素
<i>random_shuffle()</i>	采用均匀分布随机移动元素

每一个好的设计都显示出它的设计者个人的特质和兴趣。标准库里的容器和算法很清晰地反应了对经典数据结构和算法设计的密切关注。标准库不仅提供了从本质上来说是每个程序员都需要的最小的容器和算法集合，还包含了许多用于提供这些算法的工具，它们也可以用于扩充这个库，使之能超越这个最小集合。

这里要强调的并不是算法的设计，而是最简单最明显的算法的使用。有关算法设计与分析方面的信息，你应该去其他地方找（例如，[Knuth, 1968] 和 [Tarjar, 1983]）。与此相反，本章只打算列出标准库所提供的这些算法，并解释它们是如何在C++ 里表述的。这种关注能使那些对算法有所理解的人去更好地使用这个库，并使他们能按照构筑这个库的精神去扩充它。

标准库提供了许多有关排序、检索和基于某种次序调整序列的操作：

序列排序 (18.7节) <algorithm>	
<i>sort()</i>	以很好的平均效率排序
<i>stable_sort()</i>	排序，且维持相同元素原有的顺序
<i>partial_sort()</i>	将序列的前一部分排好序
<i>partial_sort_copy()</i>	复制的同时将序列的前一部分排好序
<i>nth_element()</i>	将第 n 个元素放到它的正确位置
<i>lower_bound()</i>	找到某个值的第一个出现
<i>upper_bound()</i>	找到大于某个值的第一个元素

序列排序 (18.7节) <algorithm> (续)	
<i>equal_range()</i>	找出具有给定值的一个子序列
<i>binary_search()</i>	在排好序的序列中确定给定元素是否存在
<i>merge()</i>	归并两个排好序的序列
<i>inplace_merge()</i>	归并两个接续的排好序的序列
<i>partition()</i>	将满足某谓词的元素都放到前面
<i>stable_partition()</i>	将满足某谓词的元素都放到前面且维持原顺序

集合算法 (18.7.5节) <algorithm>	
<i>include()</i>	如果一个序列是另一个的子序列则真
<i>set_union()</i>	构造一个已排序的并集
<i>set_intersection()</i>	构造一个已排序的交集
<i>set_difference()</i>	构造一个已排序序列, 其中包含所有在第一个序列中但不在第二个序列中的元素
<i>set_symmetric_difference()</i>	构造一个已排序序列, 其中包括所有只在两个序列之中的元素

堆操作将序列置于一种状态, 使对它的排序比较容易进行:

堆操作 (18.8节) <algorithm>	
<i>make_heap()</i>	将序列调整得能够作为堆使用
<i>push_heap()</i>	向堆中加入一个元素
<i>pop_heap()</i>	从堆中去除元素
<i>sort_heap()</i>	对堆排序

库提供了几个基于比较的选择元素的算法:

最大和最小 (18.9节) <algorithm>	
<i>min()</i>	两个值中较小的
<i>max()</i>	两个值中较大的
<i>min_element()</i>	序列中的最小元素
<i>max_element()</i>	序列中的最大元素
<i>lexicographic_compare()</i>	两个序列中按字典序第一个在前

最后, 库还提供了产生序列的排列的方法:

排列 (18.10节) <algorithm>	
<i>next_permutation()</i>	按字典序的下一个排列
<i>prev_permutation()</i>	按字典序的前一个排列

此外, <numeric> 里还提供了几个推广的数值算法 (22.6节)。

在下面的算法描述中, 模板参数的名字很重要。*In*、*Out*、*For*、*Bi*和*Ran*分别表示输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机访问迭代器 (19.2.1节)。*Pred*表示一元谓词, *BinPred*表示二元谓词 (18.8.2节), *Cmp*表示比较操作 (17.1.4.1节、18.7节), *Op*表示一元操作, *BinOp*表示二元操作 (18.4节)。在习惯上人们总是对模板参数使用较长的名字; 然而我却发现, 在有了一点有关标准库的知识后, 那些长名字只能降低可读性, 而不会

提高之。

随机迭代器可以当做双向迭代器使用，双向迭代器可以用做前向迭代器，而前向迭代器可以用做输入迭代器或输出迭代器（19.2.1节）。如果传递一个未提供所需操作的类型，那将就在模板实例化时产生一个错误（C.13.7节）。如果所提供的类型中包含了符合规定但是语义却不正确的操作，那将会导致无法预期的运行时错误（17.1.4节）。

18.3 序列和容器

一条很好的普遍性原则是，最常用的东西也应该是最短、最容易表述而且最安全的。标准库在普遍性的名义下违背了这条原则。因为对标准库来说，普遍性是不可或缺的。例如，我们可以按如下方式找出在一个表中42的前两个出现：

```
void f(list<int>& li)
{
    list<int>::iterator p = find(li.begin(), li.end(), 42);    // 第一个出现
    if (p != li.end()) {
        list<int>::iterator q = find(++p, li.end(), 42);    // 第二个出现
        // ...
    }
    // ...
}
```

如果将`find()`表述为对一个容器的操作，我们就需要另外增加一个机制去找出元素的第二个出现。更重要的是，将这种“增加的机制”推广到每个容器和每个算法上将是很困难的。与此不同，标准库算法是在序列的基础上工作的。也就是说，算法的输入采用界定了一个序列的一对迭代器表示，第一个迭代器引用该序列的第一个元素，第二个迭代器引用超过该序列最后元素一个位置的那一点（3.8节、19.2节）。这种序列被称为是“半开”的，因为它包含了前面提到的第一个元素，但不包含第二个。这种半开序列使得在许多算法的表述中不必将空序列作为特殊情况处理。

一个序列（特别是那些能够使用随机访问迭代器的序列）也经常被称为区间。对半开区间的传统数学记法是 $[first, last)$ 或者 $[first, last[$ 。重要的是，一个序列可以是一个容器中的所有元素或者是容器的子序列。进一步说，还有一些序列（例如I/O流）根本就不是容器，然而，基于序列表述的算法对它们也都能很好地工作。

18.3.1 输入序列

写出`x.begin()`、`x.end()`来表述“x的所有元素”是很常见的、恼人的，也是容易出错的。举例来说，如果使用了几个迭代器，就容易出现给某个算法提供了一对参数，但它们实际上并不构成一个序列：

```
void f(list<string>& fruit, list<string>& citrus)
{
    typedef list<string>::const_iterator LI;

    LI p1 = find(fruit.begin(), citrus.end(), "apple");    // 错误：（不同序列）
    LI p2 = find(fruit.begin(), fruit.end(), "apple");    // ok
    LI p3 = find(citrus.begin(), citrus.end(), "pear");    // ok
    LI p4 = find(p2, p3, "peach");                        // 错误：（不同序列）
    // ...
}
```

这个例子中有两处错误。第一处很明显（一旦你怀疑存在一个错误时），但编译器却不容易检查出这种错误；第二处在真实代码中就很难看出来了，即使是对于有经验的程序员也是如此。减少所用迭代器的数目有利于缓解这类问题。在这里，我要勾勒出一种处理这种问题的途径，所用方式就是将输入序列的意向做成显式的；然而，为了将有关标准算法的讨论严格保持在标准库的范围之内，在本章里展示算法时，将不使用这种显式的输入序列。

关键思想就是把以一个序列作为输入这件事情显式表述清楚。例如，

```
template<class In, class T> In find(In first, In last, const T& v)    // 标准
{
    while (first!=last && *first!=v) ++first;
    return first;
}

template<class In, class T> In find(Iseq<In> r, const T& v)        // 扩充
{
    return find(r.first, r.second, v);
}
```

一般来说，在用了一个*Iseq*参数时，重载（13.3.2节）将能选出这个输入序列版本。

很自然，输入序列应实现为迭代器的一个对偶（17.4.1.2节）：

```
template<class In> struct Iseq : public pair<In, In> {
    Iseq(In i1, In i2) : pair<In, In>(i1, i2) {}
};
```

我们可以将需要调用*find()*的第二个版本所需的*Iseq*显式地表示出来：

```
LI p = find(Iseq<LI>(fruit.begin(), fruit.end()), "apple");
```

然而，这样做甚至比原来直接调用*find()*的方式更麻烦些。用一个简单的协助函数就能摆脱这种麻烦。特别是，一个容器的*Iseq*也就是从它的*begin()*到*end()*的元素的序列：

```
template<class C> Iseq<typename C::iterator> iseq(C& c)           // 对于容器
{
    return Iseq<typename C::iterator>(c.begin(), c.end());
}
```

这就使我们能更紧凑地表述对容器使用的算法，不必再重复地写了。例如：

```
void f(list<string>& ls)
{
    list<string>::iterator p = find(ls.begin(), ls.end(), "standard");
    list<string>::iterator q = find(iseq(ls), "extension");
    // ..
}
```

很容易定义能对数组、输入流等生成*Iseq*的*iseq()*的版本（18.13[6]）。

*Iseq*带来的最重要的益处在于它将输入序列变成明显可见的。使用*iseq()*立竿见影的效果就是消除了大量将每个输入序列表述为一对迭代器的需要，而那是令人生厌的、容易出错的。

输出序列的概念也有用，但与输入序列的概念相比，它不这么简单，使用的需求也不那么迫切（18.13[7]，另见19.2.4节）。

18.4 函数对象

许多算法只使用一些迭代器和一些值在序列上操作。例如，我们可以在一个序列中借助

于`find()`找出第一个值为7的元素,像下面这样:

```
void f(list<int>& c)
{
    list<int>::iterator p = find(c.begin(), c.end(), 7);
    // ...
}
```

要想做更有趣的事情,我们会希望算法去执行一些我们提供的代码(3.8.4节)。例如,我们可能像

```
bool less_than_7(int v)
{
    return v<7;
}

void f(list<int>& c)
{
    list<int>::iterator p = find_if(c.begin(), c.end(), less_than_7);
    // ...
}
```

这样在一个序列中找出第一个值小于7的元素。存在许多明显的需要将函数作为参数传递的使用方式,如逻辑谓词、算术运算、从元素中抽取信息的操作等。为这类使用中的每一种写一个独立的函数,既不方便也不很有效。在许多情况下,对每个元素作用的函数还需要在调用之间保存一些数据,并将其作为许多这类应用的最后结果。与独立存在的函数相比,类的成员函数能更好地服务于这种需要,因为类的对象可以保存数据。此外,有关的类里还可以提供一些操作,完成初始化和对有关数据的提取工作等。

考虑如何写出一个函数或者拟函数的类,去完成求和的工作:

```
template<class T> class Sum {
    T res;
public:
    Sum(T i = 0) : res(i) { }           // 初始化
    void operator()(T x) { res += x; }   // 累加
    T result() const { return res; }     // 返回和数
};
```

很清楚,`Sum`是为那些用0初始化,并定义有`+=`的算术类型设计的。例如:

```
void f(list<double>& ld)
{
    Sum<double> s;
    s = for_each(ld.begin(), ld.end(), s);           // 对ld的每个元素调用s()
    cout << "the sum is " << s.result() << "\n";
}
```

在这里,`for_each()`(18.5.1节)将对`ld`的每个元素调用`Sum<double>::operator()(double)`,并返回作为其第三个参数的那个对象。

这种方式能够工作的关键原因是,`for_each()`实际并不假设其第三个参数就是函数,它只假设第三个参数是某种可以用合适的参数去调用的东西。具有适当定义的对象也能很好地(通常是更好地)作为一个函数使用。例如,将一个类的应用运算符在线化,就比将一个通过指针传递的函数在线化容易得多。因此,函数对象通常比常规函数执行速度更快。如果一个类的对象具有应用运算符(11.9节),我们就称它为一个拟函数对象、函子,或者简单地称为

函数对象。

18.4.1 函数对象的基类

标准库提供了许多很有用的函数对象。为了帮助写出函数对象，标准库中还提供了几个基类：

```
template <class Arg, class Res> struct unary_function {
    typedef Arg argument_type;
    typedef Res result_type;
};

template <class Arg, class Arg2, class Res> struct binary_function {
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

这些类的用途就是为参数提供标准的名字，规定返回值类型，供用户从 *unary_function* 和 *binary_function* 派生出自己的类。按照标准库的方式统一地使用这些基类能使程序员免除许多麻烦，不必再去为弄清楚这些结构为何有用而劳神费力（18.4.4.1节）。

18.4.2 谓词

谓词就是返回 *bool* 的函数对象（或者函数）。例如，*<functional>* 定义了：

```
template <class T> struct logical_not : public unary_function<T, bool> {
    bool operator() (const T& x) const { return !x; }
};

template <class T> struct less : public binary_function<T, T, bool> {
    bool operator() (const T& x, const T& y) const { return x < y; }
};
```

一元和二元谓词在算法中非常有用。举例来说，我们可以比较两个序列，寻找某个序列中的第一个不小于在另一序列里的对应元素的元素：

```
void f(vector<int>& vi, list<int>& li)
{
    typedef list<int>::iterator LI;
    typedef vector<int>::iterator VI;
    pair<VI, LI> p1 = mismatch(vi.begin(), vi.end(), li.begin(), less<int>());
    // ...
}
```

mismatch() 算法反复将其二元谓词作用于各对对应元素，直到比较失败为止。因为这里需要的是一个对象而不是一个类型，所以应该用 *less<int>()*（包括这对括号），而不是 *less<int>*。

如果需要找的不是第一个不小于另一个序列中的对应元素的元素，想找的是第一个比对应元素小的元素，那么就可以去设法寻找第一个使得与之相反的比较谓词 *greater_equal* 失败的对。这可以写为：

```
p1 = mismatch(vi.begin(), vi.end(), li.begin(), greater_equal<int>());
```

换一种方式，我们也可以将两个序列反过来放，并用 *less_equal* 去处理：

```
pair<LI, VI> p2 = mismatch(li.begin(), li.end(), vi.begin(), less_equal<int>());
```

在18.4.4.4节中我将说明如何表述谓词“不小于”。

18.4.2.1 谓词综述

在 `<functional>` 里，标准库提供了一些常用谓词：

谓词 <code><functional></code>		
<code>equal_to</code>	二元	<code>arg1 == arg2</code>
<code>not_equal_to</code>	二元	<code>arg1 != arg2</code>
<code>greater</code>	二元	<code>arg1 > arg2</code>
<code>less</code>	二元	<code>arg1 < arg2</code>
<code>greater_equal</code>	二元	<code>arg1 >= arg2</code>
<code>less_equal</code>	二元	<code>arg1 <= arg2</code>
<code>logical_and</code>	二元	<code>arg1 && arg2</code>
<code>logical_or</code>	二元	<code>arg1 arg2</code>
<code>logical_not</code>	一元	<code>!arg</code>

`less`和`logical_not`的定义已经在18.4.2节给出。

除了库中提供的谓词外，用户也可以写出自己的谓词，这种用户提供的谓词对于简单而优美地使用标准库和标准算法是不可或缺的。当我们想把算法应用于那些在设计时并没有考虑标准库和标准算法的类时，定义谓词的能力就特别重要了。举个例子，考虑10.4.6节中`Club`类的一个变形：

```
class Person { /* ... */ };

struct Club {
    string name;
    list<Person*> members;
    list<Person*> officers;
    // ...

    Club(const string& n);
};
```

用一个给定的名字在一个`list<Club>`里寻找相应的`Club`，这显然是一件很合理工作；但标准库算法`find_if()`并不知道`Club`。虽然这个库算法知道如何检测相等，但我们并不希望基于一个`Club`的整个值去找到它。相反，这里希望将`Club`的名字当做关键码使用。所以，我们需要写一个谓词来反应这一情况：

```
class Club_eq : public unary_function<Club, bool> {
    string s;
public:
    explicit Club_eq(const string& ss) : s(ss) {}
    bool operator()(const Club& c) const { return c.name==s; }
};
```

定义有用的谓词非常简单。一旦为用户定义类型定义了一个适当的谓词，对它们使用标准算法也就既简单又有效了，正如只牵涉到简单类型的那些例子一样。例如，

```
void f(list<Club>& lc)
{
    typedef list<Club>::iterator LCI;
    LCI p = find_if(lc.begin(), lc.end(), Club_eq("Dining Philosophers"));
    // ...
}
```

18.4.3 算术函数对象

在处理各种数值类时，能够把标准的算术函数作为函数对象使用常常也是很有用的。因此，标准库在 `<functional>` 里提供了：

算术运算 <functional>		
<i>plus</i>	二元	$\arg 1 + \arg 2$
<i>minus</i>	二元	$\arg 1 - \arg 2$
<i>multiplies</i>	二元	$\arg 1 * \arg 2$
<i>divides</i>	二元	$\arg 1 / \arg 2$
<i>modulus</i>	二元	$\arg 1 \% \arg 2$
<i>negate</i>	一元	$-\arg$

我们可以用 *multiplies* 去乘两个向量里的各个元素，由此生成第三个向量：

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
    transform(a.begin(), a.end(), b.begin(), back_inserter(res), multiplies<double>());
}
```

back_insert() 在19.2.4节解释。在22.6节可以找到一些数值算法。

18.4.4 约束器、适配器和否定器

我们可以使用自己写出的谓词或者函数对象，也可以依赖于标准库所提供的。然而，当我们需要某个新谓词时，经常会发现这个新谓词只是某个已有谓词的一种差异并不大的变形。标准库为支持函数对象的组合提供了：

18.4.4.1节 约束器 (binder)，通过将一个参数约束到某个值，使我们可以将两个参数的函数对象当做一个参数的函数对象使用。

18.4.4.2节 成员函数适配器，使成员函数可以被用做算法的参数。

18.4.4.3节 函数指针适配器，使函数指针可以被作为算法的参数。

18.4.4.4节 否定器 (negater)，使我们能描述某个谓词的否定。

汇集在一起，这些函数对象被统称为适配器。这些适配器具有统一的结构，它们都依赖于函数对象基类 *unary_function* 和 *binary_function* (18.4.1节)。对这些适配器中的每一个都提供了一个协助函数，它以一个函数对象为参数，返回另一个合适的函数对象。在被这个函数对象的 *operator()()* 调用时，该函数对象将能做出我们所需要的动作。也就是说，这种适配器是一种形式简单的高阶函数，它以一个函数为参数并具此产生另一个新的函数：

约束器、适配器和否定器 <functional>		
<i>bind2nd(y)</i>	<i>binder2nd</i>	以y作为第二个参数调用二元函数
<i>bind1st(x)</i>	<i>binder1st</i>	以x作为第一个参数调用二元函数
<i>mem_fun()</i>	<i>mem_fun_t</i>	通过指针调用0元成员函数
	<i>mem_fun1_t</i>	通过指针调用一元成员函数
	<i>const_mem_fun_t</i>	通过指针调用0元const成员函数
	<i>const_mem_fun1_t</i>	通过指针调用一元const成员函数

约束器、适配器和否定器 <functional> (续)

<i>mem_fun_ref()</i>	<i>mem_fun_ref_t</i>	通过引用调用0元成员函数
	<i>mem_fun1_ref_t</i>	通过引用调用一元成员函数
	<i>const_mem_fun_ref_t</i>	通过引用调用0元const成员函数
	<i>const_mem_fun1_ref_t</i>	通过引用调用一元const成员函数
<i>ptr_fun()</i>	<i>pointer_to_unary_function</i>	调用一元函数指针
<i>ptr_fun()</i>	<i>pointer_to_binary_function</i>	调用二元函数指针
<i>not1()</i>	<i>unary_negate</i>	否定一元谓词
<i>not2()</i>	<i>binary_negate</i>	否定二元谓词

18.4.4.1 约束器

像`less`这样的二元谓词(18.4.2节)非常有用,也非常灵活,然而我们很快就会发现,最有用的一类谓词是用一个固定的参数反复地去与一个容器中各个元素做比较。函数`less_than_7()`(18.4节)就是一个典型的例子。`less`操作要求我们在每次调用时都显式地提供两个参数,因此它常常不是立即可用的。这时,我们可以定义

```
template <class T> class less_than : public unary_function<T, bool> {
    T arg2;
public:
    explicit less_than(const T& x) : arg2(x) { }
    bool operator()(const T& x) const { return x<arg2; }
};
```

现在我们可以写

```
void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), less_than<int>(7));
    // ...
}
```

这里我们必须写`less_than<int>(7)`而不是`less_than(7)`,因为模板参数`<int>`无法从构造函数的参数(7)的类型推导出来(13.3.1节)。

这个`less_than`谓词具有通用性。最重要的是,我们定义它只是通过固定了,或者说是约束了`less`的第二个参数。这种通过约束一个参数而形成的组合是很常见、很有用的,也是很少令人生厌的,因此标准库提供了一个专门的标准类来支持做这一点:

```
template <class BinOp>
class binder2nd
    : public unary_function<typename BinOp::first_argument_type,
                           typename BinOp::result_type> {
protected:
    BinOp op;
    typename BinOp::second_argument_type arg2;
public:
    binder2nd(const BinOp& x, const typename BinOp::second_argument_type& v)
        : op(x), arg2(v) { }
    result_type operator()(const argument_type& x) const { return op(x, arg2); }
};

template <class BinOp, class T> binder2nd<BinOp> bind2nd(const BinOp& op, const T & V
{
```

```

    return binder2nd<BinOp>(op, v);
}

```

举例来说，我们能用**bind2nd()**从二元谓词**less**和值7出发建立起一元谓词“小于7”：

```

void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), bind2nd(less<int>(), 7));
    // ...
}

```

这样做的可读性如何？效率怎么样？对于很一般的C++实现，与18.4节中用**less_than_7()**写的原来版本相比，现在的这种写法在时间和空间上效率都更高一些。这个比较也很容易实现在线化。

有关的记法也完全符合逻辑，但确实需要一点时间去习惯它。归根到底，定义一个带约束参数的命名操作总是有价值的：

```

template <class T> struct less_than : public binder2nd< less<T> > {
    explicit less_than(const T& x) : binder2nd< less<T> >(less<T>(), x) {}
};

void f(list<int>& c)
{
    list<int>::const_iterator p = find_if(c.begin(), c.end(), less_than<int>(7));
    // ...
}

```

通过**less**而不是直接用 `<` 定义**less_than**也很重要。按照这种方式，**less_than**就能由**less**的可能存在的任何专门化中获得利益（13.5节、19.2.2节）。

与**bind2nd()**和**binder2nd**一起，`<functional>`里还提供了**bind1st()**和**binder1st**，用于约束二元函数的第一个参数。

通过约束一个参数，**bind1st()**和**bind2nd()**所履行的是一种类似人们常说的Curring化[⊖]的服务。

18.4.4.2 成员函数适配器

许多算法需要调用一个标准的或者用户定义的操作。自然，用户常常希望去调用某个成员函数。例如（3.8.5节）：

```

void draw_all(list<Shape*>& c)
{
    for_each(c.begin(), c.end(), &Shape::draw); // 呜呼！错误
}

```

问题在于，这个成员函数总需要通过一个对象去调用：**p -> mf()**。但是**for_each()**这样的函数则是通过简单的应用**f()**调用其函数对象。由于这种情况，我们就需要有一种方便而有效的方式去建立起某种东西，使一个算法能够去调用成员函数。另一种可能方式是另外做一组算法，使一个版本用于成员函数，另一个版本用于常规函数。事情还更糟些，我们还要为对象（而不是对象指针）的容器另外做一套算法版本。与约束器（18.4.4.1节）一样，这个问题可以通过一个类加一个函数的方式解决。首先考虑一种常见情况，在这里我们希望对指针容器

⊖ Curring化，指逻辑学家Curry提出的一种观点（或说是操作）。例如，将二元函数f(x, y)写成f x y，看做是将f x（函数f作用于x的结果，这里将它看做一个一元函数）作用于y。——译者注

的成员调用一个无参的成员函数：

```
template<class R, class T> class mem_fun_t : public unary_function<T*, R> {
    R (T::*pmf) ();
public:
    explicit mem_fun_t(R (T::*p)()) : pmf(p) {}
    R operator()(T* p) const { return (p->*pmf)(); } // 通过指针调用
};

template<class R, class T> mem_fun_t<R, T> mem_fun(R (T::*f)())
{
    return mem_fun_t<R, T>(f);
}
```

这样就可以处理Shape::draw()的例子了：

```
void draw_all(list<Shape*>& lsp) // 通过对象指针调用无参成员函数
{
    for_each(lsp.begin(), lsp.end(), mem_fun(&Shape::draw)); // 画出各个形状
}
```

此外，我们还需要一个类和一个mem_fun()函数来处理一个参数的成员函数。我们还需要另一些版本，去处理直接通过对象（而不是通过指针）的调用，那些函数名字是mem_fun_ref()。最后，我们还需要一些版本来处理const成员函数：

```
template<class R, class T> mem_fun_t<R, T> mem_fun(R (T::*f)());
// 为一元成员、const成员、const一元成员的版本（见18.4.4节的表）

template<class R, class T> mem_fun_ref_t<R, T> mem_fun_ref(R (T::*f)());
// 为一元成员、const成员、const一元成员的版本（见18.4.4节的表）
```

有了从<functional>来的这些成员函数适配器，我们就能写：

```
void f(list<string>& ls) // 对于对象使用无参成员函数
{
    typedef list<string>::iterator LSI;
    LSI p = find_if(ls.begin(), ls.end(), mem_fun_ref(&string::empty)); // 找到" "
}

void rotate_all(list<Shape*>& ls, int angle)
// 通过对象指针使用一个参数的成员函数
{
    for_each(ls.begin(), ls.end(), bind2nd(mem_fun(&Shape::rotate), angle));
}
```

标准库不需要去处理多于一个参数的成员函数，因为没有标准库算法以多于两个参数的函数作为操作数。

18.4.4.3 函数指针适配器

一个算法并不关心其“函数参数”是函数、函数指针或者是函数对象；然而约束器（18.4.4.1节）就需要关心，因为它需要保存一个副本供以后使用。为此，标准库在<functional>里提供了两个适配器，使函数指针也可以与标准算法一起使用。它们的定义和实现很像成员函数适配器（18.4.4.2节），这里也用了一对函数和一对类：

```
template <class A, class R> pointer_to_unary_function<A, R> ptr_fun(R (*f)(A));

template <class A, class A2, class R>
    pointer_to_binary_function<A, A2, R> ptr_fun(R (*f)(A, A2));
```

有了这些函数指针适配器，我们就可以与约束器一起使用常规函数了

```
class Record { /* ... */ };

bool name_key_eq(const Record&, const char*);    // 基于名字比较
bool ssn_key_eq(const Record&, long);           // 基于编号比较

void f(list<Record>& lr)    // 用函数指针
{
    typedef list<Record>::iterator LI;
    LI p = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(name_key_eq), "John Brown"));
    LI q = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(ssn_key_eq), 1234567890));
    // ...
}
```

这样就能从表`lr`里找出那些与关键码**John Brown**和**1234567890**匹配的元素。

18.4.4.4 否定器

谓词否定器与约束器有密切关系，因为它们也是取来一个操作，由它产生出另一个有关的操作。否定器的定义和实现也遵循成员函数适配器的模式（18.4.4.2节）。它们的定义极其简单，但这种简单性却被所用的很长的标准名字搞模糊了：

```
template <class Pred>
class unary_negate : public unary_function<typename Pred::argument_type, bool> {
    Pred op;
public:
    explicit unary_negate(const Pred& p) : op(p) {}
    bool operator()(const argument_type& x) const { return !op(x); }
};

template <class Pred>
class binary_negate : public binary_function<typename Pred::first_argument_type,
                                              typename Pred::second_argument_type, bool> {
    typedef first_argument_type Arg;
    typedef second_argument_type Arg2;
    Pred op;
public:
    explicit binary_negate(const Pred& p) : op(p) {}
    bool operator()(const Arg& x, const Arg2& y) const { return !op(x, y); }
};

template<class Pred> unary_negate<Pred> not1(const Pred& p);    // 一元否定
template<class Pred> binary_negate<Pred> not2(const Pred& p);  // 二元否定
```

这些类和函数也在 `<functional>` 里声明。名字`first_argument_type`和`second_argument_type`等来自标准基类`unary_function`和`binary_function`。

与约束器一样，最好是间接地通过其协助函数使用否定器。举个例子，我们可以表述二元谓词“不小于”，并用它找出第一对对应元素，其中的第一个元素大于等于第二个元素：

```
void f(vector<int>& vi, list<int>& li) // revised example from §18.4.2
{
    // ...
    p1 = mismatch(vi.begin(), vi.end(), li.begin(), not2(less<int>()));
    // ...
}
```

这就是让`p1`找出使得不小于谓词失败的第一对元素。

谓词处理布尔条件，这里没有与按位运算符`|`、`&`、`^`和`~`等价的东西。

当然，约束器、适配器和否定器的组合也非常有用。例如，

```
extern "C" int strcmp(const char*, const char*); // 来自 <cstdlib>

void f(list<char*> &ls) // 用函数指针
{
    typedef list<char*>::const_iterator LI;
    LI p = find_if(ls.begin(), ls.end(), not1(bind2nd(ptr_fun(strcmp), "funny")));
}
```

这是要在`ls`里找到包含C风格字符串“funny”的元素。需要否定器是因为`strcmp()`在字符串相等时返回的是0。

18.5 非修改性序列算法

非修改性序列算法是不必写循环而从序列中找出某些东西的基本工具。此外，这些算法还使我们能弄清有关元素的某些情况。这些算法都可以取`const`迭代器为参数（19.2.1节），而且，除了`for_each()`之外，都不应该用于调用修改序列中元素的操作。

18.5.1 对每个做——`for_each`

我们使用库就是想从别人的工作中获益。使用库函数、类、算法等能使我们免去许多工作，不必再去发明、设计、书写某些东西，排除其中的错误，以及为它撰写文档，等等。使用标准库也能使产生出的代码更便于其他人阅读，只要他们熟悉这个库。否则的话，人们就不得不花许多时间和精力去理解那些“家酿”的代码。

使用标准库算法还有一个重要好处，它们使程序员不必去写显式的循环。循环令人生厌，也容易出错。`for_each()`算法在某种意义上说是最简单的算法，因为它别的什么也没做，就是去掉了——一个显式的循环。`for_each()`简单地对一个序列调用其操作符参数：

```
template<class In, class Op> Op for_each(In first, In last, Op f)
{
    while (first != last) f(*first++);
    return f;
}
```

人们想以这种方式调用哪些函数呢？如果你希望积累由元素中得到的信息，那么请考虑`accumulate()`（22.6节）。如果你希望在一个序列中找出某些东西，请考虑`find()`和`find_if()`（18.5.2节）。如果你想要改变或者删除元素，请考虑`replace()`（18.6.4节）或者`remove()`（18.6.5节）。一般来说，在使用`for_each()`之前，总应该先考虑是否存在更特殊的算法能为你做更多的事情。

`for_each()`的结果就是那个作为参数的函数或者函数对象。正如在`Sum`例子（18.4节）中所示，这将使信息能够被传递给调用者。

`for_each()`的一种最常见用途就是从序列的元素中提取信息。举个例子，考虑收集一些`Club`（18.4节）的名字：

```
void extract(const list<Club> &lc, list<Person*> &off) // 将officers从lc放入off
{
    for_each(lc.begin(), lc.end(), Extract_officers(off));
}
```

与18.4节和18.4.2节的例子一样，我们要定义一个函数类来提取所需的信息。在目前这个情况下，需要提取出的那些名字可以在我们的`list<Club>`里的`list<Person*>`中找到。因此，`Extract_officers`需要将`Club`的`officers`表复制到我们的表里：

```
class Extract_officers {
    list<Person*> &lst;
public:
    explicit Extract_officers(list<Person*> &x) : lst(x) {}

    void operator()(const Club& c) const
        { copy(c.officers.begin(), c.officers.end(), back_inserter(lst)); }
};
```

我们现在就能打印出有关的名字了，还是用`for_each()`：

```
void extract_and_print(const list<Club> &lc)
{
    list<Person*> off;
    extract(lc, off);
    for_each(off.begin(), off.end(), Print_name(cout));
}
```

写出`Print_name`被留做一个练习（18.13[4]）。

`for_each()`是一个很典型的非修改性算法，因为它并不显式地修改序列。当然，如果应用到非`const`序列，`for_each()`的操作（它的第三个参数）也可以修改序列中的元素。作为一个实例，请见11.9节中`negate()`的用法。

18.5.2 查找族函数

`find()`算法从头至尾查看一个序列或者一对序列，寻找一个值或者与某个谓词匹配的元素。`find()`的最简单版本查找一个值或者与某谓词的一个匹配：

```
template<class In, class T> In find(In first, In last, const T& val);
template<class In, class Pred> In find_if(In first, In last, Pred p);
```

算法`find()`和`find_if()`将分别返回一个迭代器，该迭代器引用着与有关值或者谓词匹配的第二个元素。事实上，可以将`find()`看做是`find_if()`的一个用`==`作为谓词的版本。为什么这两个函数没有都取名为`find()`呢？原因是函数重载机制未必总能区分清楚两个参数数目相同的模板。考虑下面例子：

```
bool pred(int);

void f(vector<bool(*) (int)> &v1, vector<int> &v2)
{
    find(v1.begin(), v1.end(), pred);           // 找pred
    find_if(v2.begin(), v2.end(), pred);         // 找能使pred()返回真的int
}
```

如果`find()`和`find_if()`同名，必然会出现一些令人诧异的歧义性问题。一般说，带`_if`后缀都说明这个算法使用一个谓词。

`find_first_of()`算法在一个序列中找出第一个与另一个序列相匹配的元素：

```
template<class For, class For2>
    For find_first_of(For first, For last, For2 first2, For2 last2);
```

```
template<class For, class For2, class BinPred>
    For find_first_of(For first, For last, For2 first2, For2 last2, BinPred p);
```

例如,

```
int x[] = { 1, 3, 4 };
int y[] = { 0, 2, 3, 4, 5 };

void f()
{
    int* p = find_first_of(x, x+3, y, y+5);    // p = &x[1]
    int* q = find_first_of(p+1, x+3, y, y+5);  // q = &x[2]
}
```

指针 p 将指向 $x[1]$, 因为3是 x 里第一个与 y 匹配的元素。与此类似, q 将指向 $x[2]$ 。

`adjacent_find()` 算法寻找一对相邻的互相匹配的值:

```
template<class For> For adjacent_find(For first, For last);

template<class For, class BinPred> For adjacent_find(For first, For last, BinPred p);
```

返回值是一个迭代器, 指向找到的匹配中的第一个元素。例如,

```
void f(vector<string>& text)
{
    vector<string>::iterator p = adjacent_find(text.begin(), text.end());
    if (p != text.end() && *p == "the") { // "the" 出现重复
        text.erase(p);
        // ...
    }
}
```

18.5.3 计数

`count()` 和 `count_if()` 算法在一个序列中统计某个值出现的次数:

```
template<class In, class T>
    typename iterator_traits<In>::difference_type count(In first, In last, const T& val);

template<class In, class Pred>
    typename iterator_traits<In>::difference_type count_if(In first, In last, Pred p);
```

`count()` 的返回类型很有趣。请看下面这个很明显的但又有点考虑不周的 `count()` 版本:

```
template<class In, class T> int count(In first, In last, const T& val)
{
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

问题是, `int` 可能不是作为结果的正确类型。在某种具有小 `int` 的机器上, 序列里有可能存在太多的元素需要 `count()`, 其数目可能无法放入 `int` 中。反过来说, 在一种特殊机器上的某个高性能实现或许希望用 `short` 来保存计数值。

事情很清楚, 在一序列中的元素个数不可能大于其迭代器之间的最大差值 (19.2.1节)。由此, 解决这个问题的第一个想法就是将返回类型定义为:

```
typename In::difference_type
```

然而，标准算法除了要用于标准容器外，还应该能用于内部数组。例如，

```
void f(char* p, int size)
{
    int n = count(p, p+size, 'e'); // 统计字符e出现的次数
    // ...
}
```

不幸的是，`char*::difference_type`在C++ 里不合法。这个问题是通过`iterator_traits`的一个部分专门化解决的（19.2.2节）。

18.5.4 相等和不匹配

`equal()` 和 `mismatch()` 算法比较两个序列：

```
template<class In, class In2> bool equal(In first, In last, In2 first2);
template<class In, class In2, class BinPred>
    bool equal(In first, In last, In2 first2, BinPred p);
template<class In, class In2> pair<In, In2> mismatch(In first, In last, In2 first2);
template<class In, class In2, class BinPred>
    pair<In, In2> mismatch(In first, In last, In2 first2, BinPred p);
```

`equal()` 算法简单地告诉我们两个序列中对应各对元素的比较是否都相等；`mismatch()` 则寻找第一对没通过比较的元素，并返回引用它们的迭代器。对于第二个序列不需要说明结尾，即没有`last2`。这里实际上假定了，在第二个序列中至少有与第一个序列一样多的元素，是把`first2 + (last - first)` 用做`last2`。在标准库里，在所有面向成对元素的使用了一对序列的操作里，始终贯彻着这种技术。

如18.5.1节所示，这两个算法也比它们初看起来更为有用，因为可以为它们提供谓词，以定义相等和匹配的含义。

请注意，两个序列不必是同一类型的。例如，

```
void f(list<int>& li, vector<double>& vd)
{
    bool b = equal(li.begin(), li.end(), vd.begin());
}
```

全部要求就是有关的谓词可以接受这两种元素。

`mismatch()` 的两个版本只在所用谓词方面有所不同。实际上，我们原本也可以将它们实现为一个带有默认模板参数的函数：

```
template<class In, class In2, class BinPred>
pair<In, In2> mismatch(In first, In last, In2 first2,
    BinPred p = equal_to<typename In::value_type>()) // 18.4.2.1节
{
    while (first != last && p(*first, *first2)) {
        ++first;
        ++first2;
    }
    return pair<In, In2>(first, first2);
}
```

采用两个函数与采用一个带有默认参数的函数之间还是有差异，在使用函数指针时就可以看

到这种差异了。无论如何，把标准算法的多个版本简单地看做是“带默认谓词的版本”，大约可以使需要记忆的模板函数的数目减少一半。

18.5.5 搜索

`search()`、`search_n()` 和 `find_end()` 算法用于寻找一序列在另一个序列中作为子序列的出现：

```
template<class For, class For2>
    For search(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For search(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class For2>
    For find_end(For first, For last, For2 first2, For2 last2);

template<class For, class For2, class BinPred>
    For find_end(For first, For last, For2 first2, For2 last2, BinPred p);

template<class For, class Size, class T>
    For search_n(For first, For last, Size n, const T& val);

template<class For, class Size, class T, class BinPred>
    For search_n(For first, For last, Size n, const T& val, BinPred p);
```

`search()` 算法在其第一个参数里找第二个序列（作为它的子序列）。如果找到了第二个序列，它就返回一个迭代器，指向位于第一个序列中的第一个匹配元素。返回序列结尾（`last`）表示“没找到”。这样，`find()` 的返回值总位于 `[first, last]` 序列中。例如：

```
string quote("Why waste time learning, when ignorance is instantaneous?");

bool in_quote(const string& s)
{
    typedef string::const_iterator SCI;
    SCI p = search(quote.begin(), quote.end(), s.begin(), s.end()); // 在quote里找到s
    return p != quote.end();
}

void g()
{
    bool b1 = in_quote("learning"); // b1 = true
    bool b2 = in_quote("lemming"); // b2 = false
}
```

可见，`search()` 就是一个将寻找子串的工作推广到所有序列的操作。这也意味着 `search()` 是一个很有用的算法。

`find_end()` 算法在它的第一个输入序列中查找作为子序列的第二个序列。如果能找到第二个序列，`find_end()` 返回的迭代器将指向第一个序列里的最后一个匹配。换句话说，`find_end()` 也就是反向的 `search()`，它在自己的第一个输入序列里找第二个序列的最后一个出现，而不是第一个出现。

`search_n()` 算法在一个序列中，对它的 `value` 参数找出至少 `n` 个连续匹配的一个序列，它返回一个迭代器，指向序列中第 `n` 个匹配的最后一个元素。

18.6 修改性序列算法

如果你希望修改一个序列，你当然可以显式地写一个迭代穿过它，这样就可以修改其中

的值了。当然，只要有可能，我们都宁愿避免这类程序设计，欢迎更简单更具有系统化风格的程序设计。可作为替代的方式就是采用那些能遍历序列、执行特定工作的算法。在只需读其中的值时，非修改性算法（18.5节）就能服务于我们的需要了。提供修改性算法则是为完成更一般形式的更新，有些算法更新序列，另一些则在遍历过程中，基于所发现的信息生成新的序列。

标准算法通过迭代器在数据结构上工作，这就意味着，要在序列中插入一个元素或者删除一个元素都很不容易。例如，只有一个迭代器，我们怎能找到需要从中删除被指向的元素的那个容器呢？除非用的是特殊的迭代器（例如，插入器，3.8节、19.2.4节），借助于迭代器的操作都不能改变容器的大小。除了无法插入或删除元素以外，这些算法可以修改元素的值、交换元素、复制元素等。甚至`remove()`的操作也是通过复写掉一些元素的方式完成的（18.6.5节）。总而言之，基本的修改操作产生的输出是其输入序列的修改后的副本。那些看起来真正是在修改序列的算法，实际上只是一些变形，它们在序列内做复制。

18.6.1 复制

复制是由一个序列产生另一个序列的最简单方式。基本复制操作的定义极其简单：

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first != last) *res++ = *first++;
    return res;
}

template<class Bi, class Bi2> Bi2 copy_backward(Bi first, Bi last, Bi2 res)
{
    while (first != last) *--res = *--last;
    return res;
}
```

复制算法的目标不必是容器。任何可以用输出迭代器描述的东西（19.2.6节）都可以用。例如：

```
void f(list<Club>& lc, ostream& os)
{
    copy(lc.begin(), lc.end(), ostream_iterator<Club>(os));
}
```

要从一个序列读数据，我们就需要描述这个序列在何处开始和结束。要向一个序列写，有一个描述向哪里写的迭代器也就够了。当然，我们必须注意不要写过了目标的末端，保证不会出错的一种方式是采用插入器（19.2.4节），使目标可以根据需要增长。例如，

```
void f(const vector<char>& vs, vector<char>& v)
{
    copy(vs.begin(), vs.end(), v.begin()); // 可能写过v的末端
    copy(vs.begin(), vs.end(), back_inserter(v)); // 在v的末端增加元素
}
```

输入序列和输出序列之间有可能出现重叠。在序列不出现重叠时或者当输出序列的末端位于输入序列里时，我们就可以采用`copy()`。当输出序列的开始端位于输入序列里时，我们则应该用`copy_backward()`。按照这种方式，直到复制之时元素都不会被覆盖掉。另见18.13[13]。

很自然，如果想做反向的复制，我们就需要输入和输出序列都有双向迭代器（19.2.1节）。例如，


```

void f(vector<char>& vc)
{
    copy_backward(vc.begin(), vc.end(), ostream_iterator<char>(cout)); // 错误
    vector<char> v(vc.size());
    copy_backward(vc.begin(), vc.end(), v.end()); // ok
    copy(v.begin(), v.end(), ostream_iterator<char>(cout)); // ok
}

```

我们常常只需要复制那些满足某种准则的元素。不幸的是，算法`copy_if()`被从标准库提供的算法集合中抛弃了（是我之过）。但另一方面，定义它也极其简单：

```

template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
{
    while (first != last) {
        if (p(*first)) *res++ = *first;
        ++first;
    }
    return res;
}

```

现在，如果我们想打印出所有值大于 n 的元素，就可以像下面这样做：

```

void f(list<int>&ld, int n, ostream& os)
{
    copy_if(ld.begin(), ld.end(), ostream_iterator<int>(os), bind2nd(greater<int>(), n));
}

```

另见`remove_copy_if()`（18.6.5节）。

18.6.2 变换

情况可能容易把人搞糊涂，`transform()`并不修改它的输入，实际上，它是基于用户提供的操作对输入做某种变换（transform），生成出一个输出序列：

```

template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op)
{
    while (first != last) *res++ = op(*first++);
    return res;
}

template<class In, class In2, class Out, class BinOp>
Out transform(In first, In last, In2 first2, Out res, BinOp op)
{
    while (first != last) *res++ = op(*first++, *first2++);
    return res;
}

```

`transform()`在读一个序列并产生输出方面很像`copy()`，但它不是直接写出各个元素，而是写出对元素操作的结果。这样，我们实际上可以将`copy()`定义为一个`transform()`。只要用一个直接返回其参数值的操作：

```

template<class T> T identity(const T& x) { return x; }

template<class In, class Out> Out ccopy(In first, In last, Out res)
{

```

```
return transform(first, last, res, identity<typename iterator_traits<In>::value_type>());
}
```

这里必须对`identity`采用显式限定，以便能从函数模板得到特定的函数。用`iterator_traits`模板（19.2.2节）是为了取得`In`的成员类型。

认识`transform()`还可以采用另一种观点，那就是将它看成`for_each`的一种显式生成输出的变形。例如，我们可以利用`transform()`从一个`Club`的表生成一个名字`string`的表：

```
string nameof(const Club& c) { return c.name; } // 提取名字串

void f(list<Club>& lc)
{
    transform(lc.begin(), lc.end(), ostream_iterator<string>(cout), nameof);
}
```

将`transform()`称为“变换”的一个原因是，人们常常将操作的结果写回到参数的出处。考虑删除由一组指针所指的一些对象：

```
struct Delete_ptr { // 利用函数对象取得在线效果
    template<class T> T* operator()(T* p) { delete p; return 0; }
};

void purge(deque<Shape*>& s)
{
    transform(s.begin(), s.end(), s.begin(), Delete_ptr());
}
```

`transform()`算法总要产生输出序列。在这里，我将这种输出送回输入序列，使`Delete_ptr()(p)`的效果是`p = Delete_ptr()(p)`。这也就是我选择让`Delete_ptr::operator()()`返回0的原因。

以两个序列为输入的`transform()`使人能够组合起出自两个来源的信息。例如，在一个动画里可能有一个例行程序，它通过应用一个变换去更新存储在表中的一些形状的位置：

```
Shape* move_shape(Shape* s, Point p) // *s += p
{
    s->move_to(s->center()+p);
    return s;
}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    // 在对应对象上调用操作；
    transform(ls.begin(), ls.end(), oper.begin(), ls.begin(), move_shape);
}
```

我原本并不想让`move_shape()`返回一个值，但是，`transform()`坚持要求用它的操作的结果做赋值，因此我只好让`move_shape()`返回其第一个操作数，使之可以被写回原处。

有时我们没有这样做的自由。举例来说，某个不是我写的、我也不想去修改的操作可能根本不返回值。有时候，这个输入序列本身是`const`。对于这些情况，我们就可能需要按照两个序列的`transform()`的方式，定义出对两个序列的`for_each()`：

```
template<class In, class In2, class BinOp>
BinOp for_each(In first, In last, In2 first2, BinOp op)
{
    while (first != last) op(*first++, *first2++);
    return op;
}
```

```

}

void update_positions(list<Shape*>& ls, vector<Point>& oper)
{
    for_each(ls.begin(), ls.end(), oper.begin(), move_shape);
}

```

在另一些时候，有一个实际上什么也不写的输出迭代器也很有用处（19.6[2]）。

不存在同时读入三个或者更多序列的标准库算法。但这种算法很容易自己写出来，或者你可以重复地使用`transform()`。

18.6.3 惟一化

在收集信息的过程中有可能出现重复，`unique()`和`unique_copy()`算法能够删除相邻出现的重复值：

```

template<class For> For unique(For first, For last);
template<class For, class BinPred> For unique(For first, For last, BinPred p);

template<class In, class Out> Out unique_copy(In first, In last, Out res);
template<class In, class Out, class BinPred>
    Out unique_copy(In first, In last, Out res, BinPred p);

```

`unique()`算法从一个序列里删除重复的元素，而`unique_copy()`则是建立一个无重复的副本。例如，

```

void f(list<string>& ls, vector<string>& vs)
{
    ls.sort(); // 表排序（17.2.2.1节）
    unique_copy(ls.begin(), ls.end(), back_inserter(vs));
}

```

这就把`ls`复制到`vs`了，并在这个过程中删除了重复元素。在这里，需要用`sort()`将同样的`string`弄到相邻的位置。

与其他标准算法一样，`unique()`也是在迭代器上操作。它没办法弄清这些迭代器所指的容器的类型，所以它也无法修改容器，而只能修改元素的值。这也就意味着，`unique()`并不是像我们设想的那样从输入序列中删除重复。实际上，它不过是把不重复的元素移向序列的前部（头部），返回指向这一段无重复子序列末端的迭代器：

```

template <class For> For unique(For first, For last)
{
    first = adjacent_find(first, last); // 18.5.2节
    return unique_copy(first, last, first);
}

```

在这段无重复子序列之后的元素留在原处没动。由此可见，这样不能删除`vector`里的重复：

```

void f(vector<string>& vs) // 警告：错误代码！
{
    sort(vs.begin(), vs.end()); // 向量排序
    unique(vs.begin(), vs.end()); // 删除重复（不，没完成！）
}

```

事实上，在通过将一个序列的后面元素前移的方式去除重复地过程里，`unique()`还可能引进新的重复。例如，

```
int main()
{
    char v[] = "abbcccdde";
    char* p = unique(v, v+strlen(v));
    cout << v << " " << p-v << "\n";
}
```

产生的是

```
abcdecde 5
```

这时`p`将指向第二个`c`。

被认为能删除元素（实际不能）的算法都有两种形式。一种“普通”形式的都以`unique()`这样的方式重新排列元素，另一种则以`unique_copy()`的同样方式生成一个新序列。在这里用`_copy`后缀区分这两类算法。

要真正从一个容器里删除重复元素，我们就必须显式地收缩这个容器：

```
template<class C> void eliminate_duplicates(C& c)
{
    sort(c.begin(), c.end());           // 排序
    typename C::iterator p = unique(c.begin(), c.end()); // 压紧
    c.erase(p, c.end());                 // 收缩
}
```

请注意，`eliminate_duplicates()`对内部数组没有意义，然而`unique()`却可以作用于数组。

在3.8.3节可以找到一个有关`unique_copy()`的例子。

18.6.3.1 排序准则

为了清除所有的重复，输入序列必须首先排序（18.7.1节）。`unique()`和`unique_copy()`都用`==`作为默认比较准则，但也允许用户提供另外的准则。例如，我们可以修改18.5.1节的例子，清除重复的名字。在抽取出所有`Club`的官员名字之后，我们将得到一个称为`off`的`list<Person*>`（18.5.1节）。随后就可以用

```
eliminate_duplicates(off);
```

这样方式删除重复了。但是，这样做实际上依赖于指针排序，且假定了每个指针惟一地标识了一个人。一般说，我们应该检查`Person`记录，以确定是否应该认为它们相同。我们可以写：

```
bool operator==(const Person& x, const Person& y) // 对象相等
{
    // 比较x和y是否相等
}

bool operator<(const Person& x, const Person& y) // 对象小于
{
    // 比较x和y的顺序
}

bool Person_eq(const Person* x, const Person* y) // 通过指针的相等
{
    return *x == *y;
}

bool Person_lt(const Person* x, const Person* y) // 通过指针的小于
{
    return *x < *y;
}
```

```

void extract_and_print(const list<Club>& lc)
{
    list<Person*> off;
    extract(lc, off);
    off.sort(off, Person_lt);
    list<Club>::iterator p = unique(off.begin(), off.end(), Person_eq);
    for_each(off.begin(), p, Print_name(cout));
}

```

应该确认在排序和删除重复时使用的是同一个准则。< 和 == 的指针比较有默认的意义，而将这个意义用于作为被指对象的比较准则几乎是毫无用处的。

18.6.4 取代

replace() 算法遍历一个序列，将其中的一些值用某些给定的值取代。它们也遵循 *find/find_if* 和 *unique/unique_copy* 给出了轮廓的操作模型，这样就产生了总共4个变形。同样，这些代码也简单得很，可以展示如下：

```

template<class For, class T>
void replace(For first, For last, const T& val, const T& new_val)
{
    while (first != last) {
        if (*first == val) *first = new_val;
        ++first;
    }
}

template<class For, class Pred, class T>
void replace_if(For first, For last, Pred p, const T& new_val)
{
    while (first != last) {
        if (p(*first)) *first = new_val;
        ++first;
    }
}

template<class In, class Out, class T>
Out replace_copy(In first, In last, Out res, const T& val, const T& new_val)
{
    while (first != last) {
        *res++ = (*first == val) ? new_val : *first;
        ++first;
    }
    return res;
}

template<class In, class Out, class Pred, class T>
Out replace_copy_if(In first, In last, Out res, Pred p, const T& new_val)
{
    while (first != last) {
        *res++ = p(*first) ? new_val : *first;
        ++first;
    }
    return res;
}

```

我们可能会想遍历一个 *string* 的表，将我的家乡名字的常见英语翻译 Aarhus 用它正确的名字

Århus取代:

```
void f(list<string>& towns)
{
    replace(towns.begin(), towns.end(), "Aarhus", "Århus");
}
```

这样做需要依赖于一个扩充的字符集 (C.3.3节)。

18.6.5 删除

remove() 基于一个值或者一个谓词, 删除 (**remove**) 一个序列中的元素:

```
template<class For, class T> For remove(For first, For last, const T& val);
template<class For, class Pred> For remove_if(For first, For last, Pred p);
template<class In, class Out, class T>
    Out remove_copy(In first, In last, Out res, const T& val);
template<class In, class Out, class Pred>
    Out remove_copy_if(In first, In last, Out res, Pred p);
```

假定 **Club** 有一个地址, 我们可以如下产生出一个位于哥本哈根的 **Club** 的表:

```
class located_in : public unary_function< Club, bool > {
    string town;
public:
    located_in(const string& ss) : town(ss) {}
    bool operator()(const Club& c) const { return c.town == town; }
};

void f(list<Club>& lc)
{
    remove_copy_if(lc.begin(), lc.end(),
        ostream_iterator<Club>(cout), not1(located_in("København")));
}
```

可见, **remove_copy_if()** 也就是带着否定条件的 **copy_if()** (18.6.1节), 也就是说, 一个元素要能被 **remove_copy_if()** 放到输出, 其条件就是它不与谓词匹配。

“简单的” **remove()** 把不匹配的元素压缩到序列前面, 返回指向这个压缩后的序列末端的迭代器 (另见18.6.3节)。

18.6.6 填充和生成

fill() 和 **generate()** 算法用于系统化地为序列赋值:

```
template<class For, class T> void fill(For first, For last, const T& val);
template<class Out, class Size, class T> void fill_n(Out res, Size n, const T& val);

template<class For, class Gen> void generate(For first, For last, Gen g);
template<class Out, class Size, class Gen> void generate_n(Out res, Size n, Gen g);
```

fill() 算法反复赋一个特定的值, **generate()** 算法赋的值则是通过反复调用它的函数参数得到的。可见, **fill()** 不过是 **generate()** 的一个特殊情况, 其中的生成函数反复给出同样的值。**_n** 版本给序列中前 **n** 个元素赋值。

下面的例子中使用了取自22.7节的随机数生成器 **Randint** 和 **Urand**:

```

int v1[900];
int v2[900];
vector v3;

void f()
{
    fill(v1, &v1[900], 99);           // 将v1的所有元素置为99
    generate(v2, &v2[900], Randint()); // 设置随机值 (22.7节)

    // 输出200个位于区间 [0..99] 中的随机值
    generate_n(ostream_iterator<int>(cout), 200, Urand(100)); // 见22.7节

    fill_n(back_inserter(v3), 20, 99); // 给v3加进20个值为99的元素
}

```

`generate()` 和 `fill()` 都是做赋值，而不是做初始化。如果你需要操作原始存储，比如说，要将一块存储区转化为一些具有良好定义的类型和状态的对象，那就必须用取自 `<memory>` (19.4.4节) 的算法 (例如 `uninitialized_fill()`)，而不能来自 `<algorithm>` 的算法。

18.6.7 反转和旋转

偶然我们也需要重排序列里的元素：

```

template<class Bi> void reverse(Bi first, Bi last);
template<class Bi, class Out> Out reverse_copy(Bi first, Bi last, Out res);

template<class For> void rotate(For first, For middle, For last);
template<class For, class Out> Out rotate_copy(For first, For middle, For last, Out res);

template<class Ran> void random_shuffle(Ran first, Ran last);
template<class Ran, class Gen> void random_shuffle(Ran first, Ran last, Gen& g);

```

`reverse()` 算法反转 (`reverse`) 元素的顺序，使第一个元素变成最后一个，依次类推。
`reverse_copy()` 算法产生出一个与其输入反序的副本。

`rotate()` 算法将它的序列 `[first, last[` 看做是一个环，它旋转 (`rotate`) 这些元素，直至其中原来的 `middle` 元素到达了其 `first` 元素原来的位置。也就是说，使得原来在 `first + i` 的元素移到位置 `first + (i + (last - middle)) % (last - first)`。正是这个 `%` (取模) 运算使元素按照一个环的方式旋转，而不是简单地向左移。例如，

```

void f()
{
    string v[] = { "Frog", "and", "Peach" };

    reverse(v, v+3);           // Peach and Frog
    rotate(v, v+1, v+3);       // and Frog Peach
}

```

`rotate_copy()` 算法产生其输入的一个旋转后的副本。

按默认规定，`random_shuffle()` 用一个均匀分布的随机数生成器搅乱序列中元素的位置。也就是说，它选择序列中元素的一种排列，选择的方式是使每种排列都有同样的机会被选中。如果需要另一种分布，或者另一个更好的随机数生成器，你就可以给它提供一个。例如，采用22.7节的 `Urand` 生成器，我们可以像下面这样洗一叠扑克牌：

```

void f(deque<Card>& dc)
{
    Urand r(52);
}

```

```

    random_shuffle(dc.begin(), dc.end(), r);
    // ...
}

```

由`rotate()`等所做的元素移动都通过`swap()`完成(18.6.8节)。

18.6.8 交换

要想对容器里的元素做任何有意思的事，我们都需要将它们移来移去。这种移动最好是用`swap()`表述，这样做最简单也最高效：

```

template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}

template<class For, class For2> void iter_swap(For x, For2 y);

template<class For, class For2> For2 swap_ranges(For first, For last, For2 first2)
{
    while (first != last) iter_swap(first++, first2++);
    return first2;
}

```

为交换(`swap`)元素值需要用临时变量，也存在一些在特殊情况下能去掉临时量的技巧，但最好是避免它们，更注重简单和一目了然。`swap()`算法也对一些重要类型做了专门化，如果这种专门化很要紧的话(16.3.9节、13.5.2节)。

`iter_swap()`算法交换由其迭代器参数指向的元素。

`swap_ranges()`算法交换两个区间中的所有元素。

18.7 排序的序列

一旦我们收集好了一些数据，而后就常常需要将它们排序。一旦序列排好序，我们用某种方便方式操作这些数据的机会就会显著地增加。

要想对一个序列排序，一定要有一种比较元素的方法，这是通过二元谓词实现的(18.4.2节)。默认比较是`less`(18.4.2节)，它转而默认地使用`<`。

18.7.1 排序

`sort()`算法需要随机访问迭代器(19.2.1节)。也就是说，它们最好是用于`vector`(16.3节)和类似容器：

```

template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp> void sort(Ran first, Ran last, Cmp cmp);

template<class Ran> void stable_sort(Ran first, Ran last);
template<class Ran, class Cmp> void stable_sort(Ran first, Ran last, Cmp cmp);

```

标准`list`(17.2.2节)没有提供随机访问迭代器，所以，对`list`的排序应该用特定的`list`操作(17.2.2.1节)。

基本的`sort()`效率很高，平均为 $N \cdot \log(N)$ ，但是它的最坏情况性能却很糟，是 $O(N \cdot N)$ 。

幸运的是，这种最坏情况很罕见。如果保证在最坏情况下的性能非常重要，或者要求稳定的排序，那么就可以用`stable_sort()`，这是一种 $N \cdot \log(N) \cdot \log(N)$ 的算法，当系统中有充分的额外存储时，其性能也可以改善，趋向于 $N \cdot \log(N)$ 。对于比较起来相等的元素，`stable_sort()`能保持其相对顺序不改变，而`sort()`则不行。

有时我们只需要对序列中前面的一部分元素排序。在这种情况下，只将序列前部所需的那一部分元素排好序也是有意义的，这就是部分排序：

```
template<class Ran> void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Cmp>
    void partial_sort(Ran first, Ran middle, Ran last, Cmp cmp);

template<class In, class Ran>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2);
template<class In, class Ran, class Cmp>
    Ran partial_sort_copy(In first, In last, Ran first2, Ran last2, Cmp cmp);
```

简单的`partial_sort()`只将位于区间`first`到`middle`的元素排好序。`partial_sort_copy()`算法产生出 N 个元素，这里的 N 是输出序列中元素个数和输入序列中元素个数中较小的那一个。我们需要给定结果序列的开始和结束，因为这样才能确定我们要求排序的元素个数。例如，

```
class Compare_copies_sold {
public:
    int operator()(const Book& b1, const Book& b2) const
        { return b1.copies_sold() > b2.copies_sold(); } // 按递减排序
};

void f(const vector<Book>& sales) // 找出最大的10个
{
    vector<Book> bestsellers(10);
    partial_sort_copy(sales.begin(), sales.end(),
                     bestsellers.begin(), bestsellers.end(), Compare_copies_sold());
    copy(bestsellers.begin(), bestsellers.end(), ostream_iterator<Book>(cout, "\n"));
}
```

因为`partial_sort_copy()`的目标也必须是随机访问迭代器，我们无法直接对`cout`排序。

最后，还提供了只做必要排序的算法，一直做到将第 N 个元素放到正确位置，且保证序列中比第 N 个元素小的元素不出现在它之后：

```
template<class Ran> void nth_element(Ran first, Ran nth, Ran last);
template<class Ran, class Cmp> void nth_element(Ran first, Ran nth, Ran last, Cmp cmp);
```

这个算法对一些入特别有用，如经济学家、社会学家、教师等，他们常常需要找出中值、百分位数等。

18.7.2 二分检索

像`find()`（18.5.2节）那样的顺序检索算法对于大序列而言极其低效，但是如果没有排序或散列（17.6节），我们最多也就只能做到那样了。然而，一旦序列排好了序，我们就可以采用二分检索（binary search）方法去确定某个值是否在序列中：

```
template<class For, class T> bool binary_search(For first, For last, const T& val);
template<class For, class T, class Cmp>
    bool binary_search(For first, For last, const T& value, Cmp cmp);
```

例如:

```
void f(list<int>& c)
{
    if (binary_search(c.begin(), c.end(), 7)) { // 7在c里吗?
        // ...
    }
    // ...
}
```

`binary_search()` 返回一个 *bool* 值说明这个值是否存在。就像对 `find()` 一样, 我们常常也希望知道具有这个值的元素在序列中的位置。但是在一个序列里可能存在许多具有给定值的元素, 我们常常需要或者是找到第一个这种元素, 或者是找到所有的这种元素。因此, 标准库提供了 `equal_range()` 算法去确定相同元素的区间, 并提供了找到这种区间 `lower_bound()` 和 `upper_bound()` 的算法:

```
template<class For, class T> For lower_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For lower_bound(For first, For last, const T& val, Cmp cmp);
template<class For, class T> For upper_bound(For first, For last, const T& val);
template<class For, class T, class Cmp>
    For upper_bound(For first, For last, const T& val, Cmp cmp);

template<class For, class T> pair<For, For> equal_range(For first, For last, const T& val);
template<class For, class T, class Cmp>
    pair<For, For> equal_range(For first, For last, const T& val, Cmp cmp);
```

这些算法对应于 `multimap` (17.4.2节) 中的操作。我们可以将 `lower_bound()` 看做针对排序的序列的一种快速的 `find()` 或者 `find_if()`。例如,

```
void g(vector<int>& c)
{
    typedef vector<int>::iterator VI;
    VI p = find(c.begin(), c.end(), 7); // 可能慢:  $O(N)$ , c 不必排序
    VI q = lower_bound(c.begin(), c.end(), 7); // 可能快:  $O(\log(N))$ , c 必须排序
    // ...
}
```

如果 `lower_bound(first, last, key)` 无法找到 *k*, 它返回指向第一个大于 *k* 的元素的迭代器, 在没有这种元素时返回 `last`。 `upper_bound()` 和 `equal_range()` 也采用这种方式报告失败。这也意味着我们可以用这些算法, 去确定在哪里插入元素还能保持序列仍然为已排序的。

18.7.3 归并

有了两个已排序的序列, 我们可以用 `merge()` 将它们归并 (`merge`) 成一个新的已排序序列, 或者通过 `inplace_merge()` 归并同一个序列中的两个部分:

```
template<class In, class In2, class Out>
    Out merge(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out merge(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class Bi> void inplace_merge(Bi first, Bi middle, Bi last);
template<class Bi, class Cmp> void inplace_merge(Bi first, Bi middle, Bi last, Cmp cmp);
```

注意，这些归并算法与`list`的归并（17.2.2.1节）不同，它们并不从原来的序列中删除元素，而是做所有元素的复制。

对于那些比较起来相等的元素，来自第一个区间的元素总放在来自第二个的之前。

如果你有一个序列，它可以按照多种准则排序时，`inplace_merge()` 算法将特别有用。例如，你可能有一个按照种属（例如，鳕鱼，黑线鳕，鲱鱼等）排序的鱼类名称的`vector`，如果每个种属中的元素已经按照重量排好了序，你就可以通过应用`inplace_merge()` 去归并不同种属的信息，将整个序列按重量排好序（18.13[20]）。

18.7.4 划分

划分（`partition`）序列就是把所有满足某个谓词的元素放到不满足谓词的元素之前。标准库提供了一个`stable_partition()`，它能维持满足谓词的元素之间的以及不满足的元素之间的相对顺序。此外，标准库还提供了`partition()`，它不维持相对顺序，但在存储紧张时运行得更快一点：

```
template<class Bi, class Pred> Bi partition(Bi first, Bi last, Pred p);
template<class Bi, class Pred> Bi stable_partition(Bi first, Bi last, Pred p);
```

你可以将划分想像为根据一种特别简单的准则进行的排序。例如，

```
void f(list<Club>& lc)
{
    list<Club>::iterator p = partition(lc.begin(), lc.end(), located_in("København"));
    // ...
}
```

这样就对这个`list`做了“排序”，使位于哥本哈根的`Club`到了前面。返回值（这里的`p`）或者指向第一个不满足谓词的元素，或者指向序列结束。

18.7.5 序列上的集合运算

一个序列也可以看做一个集合。从这种观点出发，为序列提供一组集合运算（如并集和交集运算）也就有意义了。当然，除非集合是已排序的，否则这种运算将极其低效，因此标准库只提供了对排序序列的集合运算。应该特别指出的是，这些集合运算对`set`（17.4.3节）和`multiset`（17.4.4节）都能很好地工作，因为这些容器也是已排序的。

如果将这些运算应用于未排序的序列，结果序列将不一定符合平常的集合论规则。这些算法都不改变它们的输入序列，得到的输出序列仍然是已排序的。

`includes()` 算法检测第二个序列的所有元素（`[first2 : last2[`）是否都是第一个序列（`[first1 : last1[`）的元素：

```
template<class In, class In2>
bool includes(In first, In last, In2 first2, In2 last2);
template<class In, class In2, class Cmp>
bool includes(In first, In last, In2 first2, In2 last2, Cmp cmp);
```

`set_union()` 和 `set_intersection()` 都以排序序列的方式产生它们的输出：

```
template<class In, class In2, class Out>
Out set_union(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
Out set_union(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

```
template<class In, class In2, class Out>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_intersection(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

set_difference() 算法产生一个序列，其元素是第一个输入序列的成员，但不属于第二个。
set_symmetric_difference() 算法生成一个序列，其中的元素都属于两个输入序列之一但又不同同时属于两个序列：

```
template<class In, class In2, class Out>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);

template<class In, class In2, class Out>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res);
template<class In, class In2, class Out, class Cmp>
    Out set_symmetric_difference(In first, In last, In2 first2, In2 last2, Out res, Cmp cmp);
```

例如，

```
char v1[] = "abcd";
char v2[] = "cdef";

void f(char v3[])
{
    set_difference(v1, v1+4, v2, v2+4, v3);           // v3 = "ab"
    set_symmetric_difference(v1, v1+4, v2, v2+4, v3); // v3 = "abef"
}
```

18.8 堆

术语堆在不同的环境中有不同的含义。在讨论算法时，“堆”通常用于指一种组织序列元素的方式，使它的第一个元素就是具有最大值的元素；加入一个元素（用**push_heap()**）或者去除一个元素（用**pop_heap()**）都相当有效，具有最坏情况的性能 $O(\log(N))$ ，其中 N 是序列中元素的个数；对它排序（用**heap_sort()**）的最坏情况性能是 $O(N \cdot \log(N))$ 。堆通过下面一组操作实现：

```
template<class Ran> void push_heap(Ran first, Ran last);
template<class Ran, class Cmp> void push_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void pop_heap(Ran first, Ran last);
template<class Ran, class Cmp> void pop_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void make_heap(Ran first, Ran last); // 将序列转换为堆
template<class Ran, class Cmp> void make_heap(Ran first, Ran last, Cmp cmp);

template<class Ran> void sort_heap(Ran first, Ran last); // 将堆转换为序列
template<class Ran, class Cmp> void sort_heap(Ran first, Ran last, Cmp cmp);
```

堆算法的风格是独特的，实现这些功能的另一种更自然的方式是提供一个带有4个操作的适配器类。那样做的结果将产生某种类似于**priority_queue**（17.3.3节）的东西。事实上，**priority_queue**几乎可以肯定是采用堆实现的。

由**push_heap(first, last)**压入的值是 $*(last - 1)$ 。这里的假设是 $[first, last - 1[$ 已经是堆，因此**push_heap()**通过将下一个元素包括进来，使这个序列扩展为 $[first, last[$ 。这样，你就可

以从一个现存序列出发, 通过一系列的 `push_heap()` 操作构造起一个堆。与之相对应, `pop_heap(first, last)` 去掉堆的第一个元素, 方式是先将它与最后元素 (`*(last - 1)`) 交换, 而后再将 `[first, last - 1]` 做成堆。

18.9 最小和最大

这里描述的一批算法都基于元素比较选出一个值。能够找出两个元素之中最大 (`max`) 或者最小 (`min`) 者显然很有用:

```
template<class T> const T& max(const T& a, const T& b)
{
    return (a < b) ? b : a;
}

template<class T, class Cmp> const T& max(const T& a, const T& b, Cmp cmp)
{
    return (cmp(a, b)) ? b : a;
}

template<class T> const T& min(const T& a, const T& b);
template<class T, class Cmp> const T& min(const T& a, const T& b, Cmp cmp);
```

`max()` 和 `min()` 运算可以以一种很显然的方式推广, 使之能应用于序列:

```
template<class For> For max_element(For first, For last);
template<class For, class Cmp> For max_element(For first, For last, Cmp cmp);

template<class For> For min_element(For first, For last);
template<class For, class Cmp> For min_element(For first, For last, Cmp cmp);
```

最后, 很容易将字符串的字典序推广到任何带有比较操作的类型的序列:

```
template<class In, class In2>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);

template<class In, class In2, class Cmp>
bool lexicographical_compare(In first, In last, In2 first2, In2 last2, Cmp cmp)
{
    while (first != last && first2 != last2) {
        if (cmp(*first, *first2)) return true;
        if (cmp(*first2++, *first++)) return false;
    }
    return first == last && first2 != last2;
}
```

这与13.4.1节里为通用串提供的函数极其相似。当然, `lexicographical_compare()` 比较的是一般的序列而不仅是串, 它返回的是 `bool` 而不是更有用的 `int`。如果第一个序列按 `<` 比较小于第二个, 它就返回 `true`。应特别注意的是, 序列相等时返回的是 `false`。

C风格的字符串和 `string` 也是序列, 因此也可以将 `lexicographical_compare()` 用做串比较函数。例如,

```
char v1[] = "yes";
char v2[] = "no";
string s1 = "Yes";
string s2 = "No";

void f()
```

```
(
    bool b1 = lexicographical_compare(v1, v1+strlen(v1), v2, v2+strlen(v2));
    bool b2 = lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end());

    bool b3 = lexicographical_compare(v1, v1+strlen(v1), s1.begin(), s1.end());
    bool b4 = lexicographical_compare(s1.begin(), s1.end(), v1, v1+strlen(v1), Nocase());
)
```

序列不必具有相同类型——因为我们所要做的不过是比较它们的元素，而且可以提供比较的准则。这些就使`lexicographical_compare()`更具有通用性，而且也比`string`的比较操作慢一点。另见22.3.8节。

18.10 排列

对于给定的4个元素的序列，我们可以有 $4 \times 3 \times 2$ 种不同的方式对它们排列，每个这种顺序就称为一个排列。例如，对于4个字符`abcd`，我们可以产生出24个排列：

```
abcd abdc acbd acdb adbc adcb bacd badc
bcad bcda bdac bdca cabd cadb cbad cbda
cdab cdba dabc dacb dbac dbca dcab dcba
```

`next_permutation()` 和 `prev_permutation()` 函数给出一个序列的这种排列：

```
template<class Bi> bool next_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool next_permutation(Bi first, Bi last, Cmp cmp);

template<class Bi> bool prev_permutation(Bi first, Bi last);
template<class Bi, class Cmp> bool prev_permutation(Bi first, Bi last, Cmp cmp);
```

`abcd`的各种排列可以以如下方式产生：

```
int main()
{
    char v[] = "abcd";
    cout << v << '\n';
    while (next_permutation(v, v+4)) cout << v << '\n';
}
```

排列将按照字典顺序（18.9节）产生。`next_permutation()`的返回值指明下一个排列是否存在，如果不存在就返回`false`，对应序列的排列是各元素按字典顺序排列。`prev_permutation()`的返回值指明前一个排列是否存在，如果不存在时就返回`false`，对应序列中的是各元素按反字典顺序的排列。

18.11 C风格算法

C++ 标准库从C标准库继承了几个处理C风格字符串的算法，再加上一个快速排序和一个二分检索。最后这两个算法都只能用于数组。

`qsort()` 和 `bsearch()` 函数在 `<cstdlib>` 和 `<stdlib.h>` 中给出。它们都使用一个通过函数指针传递的用于比较函数，在大小各为`elem_size`的`n`个元素的数组上操作。这些元素的类型不能有用户定义的复制构造函数、复制赋值和析构函数：

```
typedef int (*__cmp)(const void*, const void*); // typedef仅为简化表示

void qsort(void* p, size_t n, size_t elem_size, __cmp); // 对p排序
void* bsearch(const void* key, void* p, size_t n, size_t elem_size, __cmp); // 在p中找key
```

`qsort()` 的用法已在7.7节介绍过。

提供这些算法完全是为了C兼容性。`sort()` (18.7.1节) 和 `search()` (18.5.5节) 是更通用的, 也应该效率更高。

18.12 忠告

- [1] 多用算法, 少用循环; 18.5节。
- [2] 在写循环时, 考虑是否它能将它表述为一个通用的算法; 18.2节。
- [3] 常规性地重温算法集合, 看看是不是能将新应用变得更明晰; 18.2节。
- [4] 保证一对迭代器参数确实表述了一个序列; 18.3.1节。
- [5] 设计时应该让使用最频繁的操作是简单而安全的; 18.3节、18.3.1节。
- [6] 把测试表述成能够作为谓词使用的形式; 18.4.2节。
- [7] 切记谓词是函数和对象, 不是类型; 18.4.2节。
- [8] 你可以用约束器从二元谓词做出一元谓词; 18.4.4.1节。
- [9] 利用 `mem_fun()` 和 `mem_fun_ref()` 将算法应用于容器; 18.4.4.2节。
- [10] 当你需要将一个参数约束到一个函数上时, 用 `ptr_fun()`; 18.4.4.3节。
- [11] 切记 `strcmp()` 用0表示“相等”, 与 `==` 不同; 18.4.4.4节。
- [12] 仅在没有更特殊的算法时, 才使用 `for_each()` 和 `transform()`; 18.5.1节。
- [13] 利用谓词, 以便能以各种比较准则和相等准则使用算法; 18.4.2.1节、18.6.3.1节。
- [14] 利用谓词和其他函数对象, 以使标准算法能用于表示范围广泛的意义; 18.4.2节。
- [15] 运算符 `<` 和 `==` 在指针上的默认意义很少适用于标准算法; 18.6.3.1节。
- [16] 算法并不直接为它们的参数序列增加或减少元素; 18.6节。
- [17] 应保证用于同一个序列的小于和相等谓词相互匹配; 18.6.3.1节。
- [18] 有时排好序的序列用起来更有效且优雅; 18.7节。
- [19] 仅为兼容性而使用 `qsort()` 和 `bsearch()`; 18.11节。

18.13 练习

对本章中的一些练习, 通过查看标准库实现的源文件可以找到有关解决方法。为你自己着想: 在查看你所用的库的实现者如何解决这些问题之前, 先试着自己去找解决方法。

- 1. (*2) 学习 $O()$ 记法。给出一个实际问题, 其中对于确定的 $N > 10$, 一个 $O(N \times N)$ 算法比另一个 $O(N)$ 算法更快。
- 2. (*2) 实现并测试四个 `mem_fun()` 和 `mem_fun_ref()` 函数 (18.4.4.2节)。
- 3. (*1) 写出一个算法 `match()`, 它很像 `mismatch()`, 但是返回指向第一对与谓词匹配元素的迭代器。
- 4. (*1.5) 实现并测试18.5.1节的 `Print_name`。
- 5. (*1) 只用标准库算法对 `list` 排序。
- 6. (*2.5) 为内部数组、`istream` 和迭代器对定义 `iseq()` 的相应版本 (18.3节)。为 `Iseq` 定义一组非修改性标准算法的适当重载。讨论怎样才能最大限度地避免歧义性, 并避免模板函数数目的爆炸性增长。
- 7. (*2) 定义与 `iseq()` 相对应的 `oseq()`。作为 `oseq()` 参数的输出序列, 应该被使用它的算法所

产生的输出取代。为你所选出的至少三种标准算法定义一组适当的重载。

8. (*1.5) 生成一个1到100的平方的`vector`，打印这些平方，对`vector`的每个元素取平方根，而后打印结果向量。
9. (*2) 写出一组函数对象，它们对参数做各种按位逻辑运算。在`char`、`int`和`bitset<67>`的向量上测试这些运算。
10. (*1) 写一个`binder3()`，它约束三元函数的第一和第二个参数以产生一个一元谓词。给出一个例子，使`binder3()`是其中的一个很有用的函数。
11. (*1.5) 写一个小程序，它删除文件里所有相邻的重复单词。提示：该程序应从句子“Write a small program that that remove adjacent repeated word from a file file”里删去that、from和file。
12. (*2.5) 为在文件中保存论文和书籍的索引定义一种记录格式。写一个程序，它能从一个标明了出版年份、作者姓名、标题中的关键字、出版商名称的文件出发写出这种记录。用户应该能要求按照某些准则对输出排序。
13. (*2) 按照`copy()`的风格实现`move()`算法。这个算法允许输入和输出序列重叠。当给了随机访问迭代器作为参数时，它能达到合理的高效率。
14. (*1.5) 生成单词`food`的所有字母重排字，即4个字母`f`、`o`、`o`、`d`的各种组合。推广这个程序，使之能以一个单词作为输入，产生这个单词的所有字母重排字。
15. (*1.5) 写一个程序，产生一个句子的所有单词重排句子，即生成该句子里所有单词的不同排列（不是所有字母的排列）。
16. (*1.5) 实现`find_if()`（18.5.2节），而后用`find_if()`实现`find()`。找一种方式做这件事，使这两个函数不需要不同的名字。
17. (*2) 实现`search()`（18.5.5节），为随机访问迭代器提供一个优化的版本。
18. (*2) 取一个排序算法（例如，你所用标准库里的`sort()`，或者13.5.2节的Shell排序），向其中插入代码，使它能在每次交换元素之后打印出正在排序的序列。
19. (*2) 不存在为双向迭代器而用的`sort()`。人们的推测是，将元素复制到`vector`而后排序，要比直接使用双向迭代器排序更快些。为双向迭代器实现一个通用排序算法，而后检验这个推测。
20. (*2.5) 设想你保存了一组钓鱼运动员的记录。对每次捕鱼保存一个有关种属、长度、重量、捕鱼日期、捕鱼人名字等的记录。按照各种准则对这些记录排序并打印出它们。提示：`inplace_merge()`。
21. (*2) 建立一些参加数学、英语、法语和生物课程的学生表，从大约40个学生中取出20名字。列出同时参加数学和英语课程的学生名单，参加法语但不参加生物与数学课程的学生名单，不参加科学课程（数学和生物）的学生名单，参加了法语和数学但不参加英语、生物课程的学生名单。
22. (*1.5) 写一个`remove()`函数，使它能真正地从容器里删除元素。

第19章 迭代器和分配器

各种数据结构与算法可以无缝地协同工作，
其原因在于……它们相互毫不知情。

——Alex Stepanov

迭代器与序列——迭代器的操作——迭代器特征类——迭代器类别——插入器——反向迭代器——流迭代器——带检查迭代器——异常和算法——分配器——标准`allocator`——用户定义分配器——低级存储函数——忠告——练习

19.1 引言

迭代器是连接容器和算法的纽带，它们为数据提供了一种抽象的观点，使写算法的人不必关心多种多样的数据结构的具体细节。反过来说，由迭代器提供一个数据访问的标准模型，也缓解了要求容器提供一组更广泛的访问操作的压力。与此类似，分配器被用于将容器的实现隔离在对存储访问的细节之外。

迭代器用对象序列作为它所支持的数据访问模型（19.2节）。分配器提供了一个映射，将具有数组和字节形式的低级数据模型映射到高级的对象模型（19.4节）。最常见的低级存储模型本身则由几个标准函数支持（19.4.4节）。

迭代器是每个程序员都需要熟悉的概念之一。相反，分配器则仅仅是一种支持机制，程序员很少需要去为它操心，很少有程序员需要去写新的分配器。

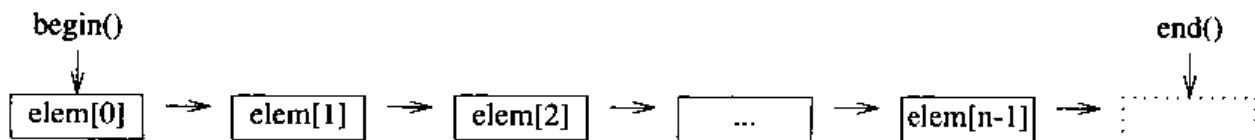
19.2 迭代器和序列

一个迭代器是一个纯的抽象，也就是说，任何在行为上像迭代器的东西也就是迭代器（3.8.2节）。迭代器是指向序列元素的指针概念的一种抽象，其最关键的属性是：

- “当前被指向的元素”（间接，用运算符 `*` 和 `->` 表示）。
- “指向下一个元素”（增量，用运算符 `++` 表示）。
- 相等（用运算符 `==` 表示）。

例如，内部类型 `int*` 就是 `int[]` 的一个迭代器，类 `list<int>::iterator` 是 `list` 类的一个迭代器。

序列也是一个抽象概念，是“我们可以从其头部开始，通过使用下一个元素操作，最终达到结束处的某种东西”：



序列的例子如数组（5.2节）、向量（16.3节）、单链表（17.8[17]）、双向链表（17.2.2节）、树（17.4.1节）、输入（21.3.1节）和输出（21.2.1节）等。其中的每一个都有它自己的适当的迭代器。

迭代器类和函数声明在名字空间`std`里，可以在 `<iterator>` 里找到。

一个迭代器并不是一个通用指针。与此相反，它只是指向数组的指针概念的一个抽象。这里不存在“空迭代器”的概念。确定一个迭代器是否指向一个元素的检测很方便，只要将它与序列的结束进行比较（而不是将它与某个空元素比较）。这一观念简化了许多算法，因为它清除了对特殊的极端情况做特别处理的需要，而且能很好地推广到任意类型的序列。

当一个迭代器指向某个元素时，我们就说它是合法的，可以对它做间接引用（通过适当地使用 `*`、`[]` 或者 `->`）。迭代器也可以是不合法的，其原因可能是它还没有初始化，或者它原来所指向的容器显式或者隐式地改变了大小（16.3.6节、16.3.8节），或者它原来指向的容器已经销毁，或者是它所指的是一个序列的结束（18.2节）。序列结束可能设想为一个虚设的元素，在序列最后元素之后的下一个位置。

19.2.1 迭代器的操作

并不是每种迭代器都支持完全相同的一组操作。例如，读入所需要的操作就与写出不同，一个 `vector` 允许方便而有效的随机访问，这种方式对于 `list` 或者 `istream` 则因为极度低效而被禁止。正因为如此，我们依据能够有效执行的操作（即只用常量时间）的情况，将迭代器分为5个类别：

	迭代器操作和类别				
类别：	输出	输入	前向	双向	随机访问
简写：	<i>Out</i>	<i>In</i>	<i>For</i>	<i>Bi</i>	<i>Ran</i>
读：		<code>= *p</code>	<code>= *p</code>	<code>= *p</code>	<code>= *p</code>
访问：		<code>-></code>	<code>-></code>	<code>-></code>	<code>-> []</code>
写：	<code>*p-</code>		<code>*p-</code>	<code>*p-</code>	<code>*p=</code>
迭代：	<code>++</code>	<code>++</code>	<code>++</code>	<code>++ --</code>	<code>++ -- + - +- -=</code>
比较：		<code>== !=</code>	<code>== !=</code>	<code>== !=</code>	<code>== != < > <= >=</code>

读和写操作都通过由 `*` 表示的迭代器间接引用：

```
*p = x;    // 通过p写x
x = *p;    // 通过p读到x里
```

一个类型要想作为迭代器类型，它就必须提供一组适当的操作。这些操作必须有它们所习用的意义，也就是说，每个操作的作用必须与它在指针上的作用相同。

各个类别的迭代器都应允许对被指对象的 `const` 和非 `const` 访问。你不能通过一个 `const` 迭代器向元素写入——无论该迭代器属于哪个类别。迭代器提供了一组操作，而对元素究竟能做什么的最终仲裁者仍然是被指向的元素的类型。

读和写都复制对象，因此，元素类型必须履行常规的复制语义（17.1.4节）。

只有随机访问迭代器可以通过加减整数，取得相对地址。当然，除了输出迭代器之外，可以得到任意两个迭代器之间的距离，因为只需通过迭代穿过一系列元素。所以，我们可以提供如下的 `distance()` 函数：

```
template<class In> typename iterator_traits<In>::difference_type distance(In first, In last)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++ != last) d++;
    return d;
}
```

对于每个迭代器`In`都定义了一个`iterator_traits<In>::difference_type`类型，用于保存元素之间的距离（19.2.2节）。

上述函数称为`distance()`而不是`operator-`，因为它可能是代价高昂的，而所有迭代器所提供的操作都具有常量的时间复杂性（17.1节）。我可不情愿无意识地对一个大序列调用这种逐个计数元素的操作。库为随机访问迭代器提供了一个更为高效的`distance()`实现。

类似地，这里还提供了`advance()`作为一种可能很慢的`+=`：

```
template <class In, class Dist> void advance(In& i, Dist n); // i+=n
```

19.2.2 迭代器特征类——`iterator_traits`

我们使用迭代器是为了获取被指向的对象和被指向的序列的信息。例如，只要给出描述某个序列的迭代器，我们就可以通过迭代器间接去操作被指向的对象，并且可以确定序列中元素的个数。为了表述这类操作，我们就必须能引用与迭代器有关的各种类型，如“被迭代器所指的对象的类型”以及“两个迭代器间距离的类型”等。与一个迭代器有关的类型都由一个`iterator_traits`模板类中的一小组声明描述：

```
template<class Iter> struct iterator_traits {
    typedef typename Iter::iterator_category iterator_category; // 19.2.3节
    typedef typename Iter::value_type value_type; // 元素类型
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::pointer pointer; // operator->() 的返回类型
    typedef typename Iter::reference reference; // operator*() 的返回类型
};
```

`difference_type`是用于表示两个迭代器之间的距离的类型，`iterator_category`是一个指明迭代器支持哪些操作的类型。针对各种常规指针，这里还特别为`<T*>`和`<const T*>`提供了专门化（13.5节）。特别地：

```
template<class T> struct iterator_traits<T*> { // 指针的专门化
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

也就是说，两个指针之间的距离用来自`<cstddef>`（6.2.1节）的标准库类型`ptrdiff_t`表示，而且指针提供的是随机访问（19.2.3节）。有了`iterator_traits`，我们就可以写出各种不依赖于迭代器参数性质的代码了。`count()`算法是个很典型的例子：

```
template<class In, class T>
typename iterator_traits<In>::difference_type count(In first, In last, const T& val)
{
    typename iterator_traits<In>::difference_type res = 0;
```

```

    while (first != last) if (*first++ == val) ++res;
    return res;
}

```

结果的类型需要借助于 `iterator_traits<In>` 表示。这样做的原因是，在这个语言里，并不存在一种方式能够通过其他类型的组合直接表达一个任意的类型。特别是无法直接表达“两个 `In` 相减的类型”。

如果不用 `iterator_traits`，我们或许能给出专门针对指针的 `count()`：

```

template<class In, class T>
typename In::difference_type count(In first, In last, const T& val);

template<class In, class T> ptrdiff_t count<T*, T>(T* first, T* last, const T& val);

```

但这至多也就是解决了 `count()` 的问题。如果我们真的想把这种技术用于数十个算法，有关距离类型的信息就会重复数十次。一般来说，最好是能将一个设计决策只在惟一的一个位置表述（23.4.2节），采用这种方式，只在这一个地方就能修改这个决策——如果真的需要。

由于每个迭代器都定义了 `iterator_traits<Iterator>`，在我们设计一个新的迭代器类型时，也就隐含地定义了一个 `iterator_traits`。如果对这个新迭代器类型，由上述通用 `iterator_traits` 默认产生的特征类不正确，我们就可以采用类似标准库为指针所用的方式，为这个迭代器类提供一个专门化。隐式生成的 `iterator_traits` 假定迭代器是一个带有 `difference_type`、`value_type` 等成员类型的类。在 `<iterator>` 里，标准库提供了一个基类型，它可以被用于定义这些成员类型：

```

template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator {
    typedef Cat iterator_category; // 19.2.3节
    typedef T value_type;         // 元素类型
    typedef Dist difference_type; // 迭代器差类型
    typedef Ptr pointer;          // -> 的返回类型
    typedef Ref reference;        // * 的返回类型
};

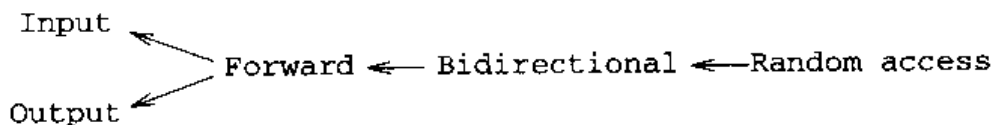
```

注意，`reference` 和 `pointer` 都不是迭代器^①。它们是计划为某些迭代器所用的，分别作为 `operator*()` 和 `operator->()` 的返回类型。

`iterator_traits` 是一个最关键的因素，它使许多依赖于迭代器的界面得到简化，使许多算法得以高效地实现。

19.2.3 迭代器类别

不同种类的迭代器（常称为迭代器类别）可以排成下面这样的层次结构：



这并不是一个类继承图，迭代器类别是基于它们所提供的操作所做的分类，许多在其他方面

① 这里作者所说的迭代器与19.2节开始处不同。在19.2节讨论的是作为抽象概念的迭代器，按照那里的说法，指针当然是一种“迭代器”。在这里讨论的则是具体的、提供了一集成员类型和其他机制的具体迭代器类，因此作者就说“指针不是迭代器”。请读者注意这种区分。——译者注

毫无关系的类型可以同属一个迭代器类别。举例来说，常规指针（19.2.2节）和**Checked_iter**（19.3节）都是随机访问迭代器。

正如在第18章中指出的，不同的算法需要以不同的迭代器作为参数。另一方面，有时同一个算法对于不同的迭代器也可能有效率不同的实现。为支持基于迭代器类别进行重载解析，标准库提供了5个类，分别代表这5个迭代器类别：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

如果去查看输入迭代器和前向迭代器所支持的操作（19.2.1节），我们或许会期望**forward_iterator_tag**是从**output_iterator_tag**以及**input_iterator_tag**派生出来的。没有这样做的原因是很模糊的，或许是不对的。但无论如何，我还是要给出一个实例，说明采用这种派生能够简化实际代码。

这些标志（tag）的继承的有用之处（仅）在于：当对几种迭代器（不是全部）可以使用同样算法时，我们可以减少对同一函数定义的版本数。考虑如何实现**distance()**：

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last);
```

存在着两种明显的不同办法：

- [1] 如果**In**是随机访问迭代器，我们可以从**last**减去**first**。
- [2] 否则我们就必须做一个迭代器的增量操作，从**first**到**last**，统计出距离来。

我们可以用一对协助函数来描述这两种选择：

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper(In first, In last, input_iterator_tag)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++ != last) d++;           // 只用增量
    return d;
}

template<class Ran>
typename iterator_traits<Ran>::difference_type
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
    return last - first;                 // 依赖于随机访问
}
```

迭代器类别标志将所期望的迭代器显式描述出来了。这种迭代器标志只用于重载解析，根本不出现在实际计算中，因此纯粹是一种编译时的选择机制。除了能自动选择协助函数外，这种技术还提供了立即的类型检查（13.2.5节）。

现在要定义能调用适当协助函数的**distance()**就非常简单了：

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last)
{
    return dist_helper(first, last, iterator_traits<In>::iterator_category());
}
```

要调用某个 `dist_helper()`，所用的 `iterator::traits<In>::iterator_category` 必须是 `input_iterator_tag` 或者 `random_iterator_tag`。无论如何，我们并不需要为前向的或双向的迭代器定义独立的 `dist_helper()` 版本。由于标志的继承性，这些情况都可以由参数取 `input_iterator_tag` 的 `dist_helper()` 处理。没有为 `output_iterator_tag` 建一个版本反应了下述事实：`distance()` 对输出迭代器毫无意义：

```
void f(vector<int>& vi,
      list<double>& ld,
      istream_iterator<string>& is1, istream_iterator<string>& is2,
      ostream_iterator<char>& os1, ostream_iterator<char>& os2)
{
    distance(vi.begin(), vi.end()); // 用减法算法
    distance(ld.begin(), ld.end()); // 用增量算法
    distance(is1, is2);             // 用增量算法
    distance(os1, os2); // 错误：迭代器类别错，dist_helper() 参数不匹配
}
```

在实际程序里，对 `istream_iterator` 调用 `distance()` 可能没有多大意思，其效果将是读输入并将它丢掉，最后返回丢掉的值的个数。

采用了 `iterator::traits<In>::iterator_category` 就使程序员可以提供不同的实现，以便那些并不关心算法实现的用户能自动为所用的每种数据结构取得最合适的实现。换句话说，它使我们能将实现细节隐藏在一个方便的界面之后。在线机制可以用来保证，这种优美效果并不需要付出运行时的效率代价。

19.2.4 插入器

将所产生的输出通过迭代器放入一个容器，意味着跟随在迭代器所指位置之后的元素就会被覆盖掉，这也意味着溢出和随之而来的存储破坏的可能性。例如，

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(), 200, 7); // 给vi[0]..vi[199] 赋值7
}
```

如果 `vi` 的元素个数少于 200，我们就有麻烦了。

在 `<iterator>` 里，标准库提供了另外三个迭代器模板类，专供处理这种问题。还加了三个函数，以使这些迭代器的使用更加方便：

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template <class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

`back_inserter()` 使元素被加在容器的末尾，`front_inserter()` 导致元素被加在前面，而“简单”`inserter()` 使元素被加在它的迭代器参数之前。对于 `inserter(c, p)`，`p` 必须是 `c` 的一个合法迭代器。当然，每次通过插入器向容器中写进元素都引起容器的增长。

在每次写入时，插入器将使用 `push_back()`、`push_front()` 或者 `insert()`（16.3.6 节）把元素插入序列中，而不是覆盖已有元素。例如，

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi), 200, 7); // 将200个7插入vi末端
}
```

插入器很有用，同时又简单而高效。例如，

```
template <class Cont>
class insert_iterator : public iterator<output_iterator_tag, void, void, void, void> {
protected:
    Cont& container;           // 要插入的容器
    typename Cont::iterator iter; // 指向容器
public:
    explicit insert_iterator(Cont& x, typename Cont::iterator i)
        : container(x), iter(i) {}

    insert_iterator& operator=(const typename Cont::value_type& val)
    {
        iter = container.insert(iter, val);
        ++iter;
        return *this;
    }

    insert_iterator& operator*() { return *this; }
    insert_iterator& operator++() { return *this; } // 前缀++
    insert_iterator operator++(int) { return *this; } // 后缀++
};
```

显然，插入器是一种输出迭代器。

insert_iterator是输出序列的一种特殊情况。我们可以与18.3.1节的**iseq**平行地定义：

```
template<class Cont>
insert_iterator<Cont>
oseq(Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
    return insert_iterator<Cont>(c, c.erase(first, last)); // 16.3.6节有erase的解释
}
```

换句话说，一个输出序列将删除所有老元素，并用输出取代它们。例如，

```
void f(list<int>& li, vector<int>& vi) // 用li的一个副本取代vi的后一半
{
    copy(li.begin(), li.end(), oseq(vi, vi.begin() + vi.size() / 2, vi.end()));
}
```

必须以容器作为**oseq**的一个参数，因为，如果只提供容器的迭代器，**oseq**将无法减小容器的规模（18.6节、18.6.3节）。

19.2.5 反向迭代器

标准容器提供了**rbegin()**和**rend()**，以便能反向地迭代通过容器的各个元素（16.3.2节）。这些成员函数返回的是**reverse_iterator**：

```
template <class Iter>
class reverse_iterator : public iterator<typename iterator_traits<Iter>::iterator_category,
                                     typename iterator_traits<Iter>::value_type,
                                     typename iterator_traits<Iter>::difference_type,
                                     typename iterator_traits<Iter>::pointer,
                                     typename iterator_traits<Iter>::reference> {
protected:
    Iter current; // current指向*this引用元素之后的那个元素
public:
    typedef Iter iterator_type;
```

```

reverse_iterator() : current() {}
explicit reverse_iterator(Iter x) : current(x) {}
template<class U> reverse_iterator(const reverse_iterator<U>& x) : current(x.base()) {}

Iter base() const { return current; } // 当前迭代器值

reference operator*() const { Iter tmp = current; return *--tmp; }
pointer operator->() const;
reference operator[] (difference_type n) const;

reverse_iterator& operator++() { --current; return *this; } // 注意：不是 ++
reverse_iterator operator++(int) { reverse_iterator t = current; --current; return t; }
reverse_iterator& operator--() { ++current; return *this; } // 注意：不是 --
reverse_iterator operator--(int) { reverse_iterator t = current; ++current; return t; }

reverse_iterator operator+(difference_type n) const;
reverse_iterator& operator+=(difference_type n);
reverse_iterator operator-(difference_type n) const;
reverse_iterator& operator-=(difference_type n);
};

```

reverse_iterator的实现中采用了一个称为**current**的**iterator**，这个**iterator**（只）能指向它的序列中的元素，再加上末端之后的一个元素。另一方面，**reverse_iterator**的“末端之后的一个”元素实际上是原序列（不可访问）的“始端之前的一个”元素。这样，为了避免访问违例，**current**总是指向**reverse_iterator**所指元素之后的一个元素。这也意味着 ***** 返回的是值 ***(current - 1)**，**++** 通过对**current**的 **--** 实现。

reverse_iterator（只）支持初始化它所用的那个迭代器所支持的操作。例如，

```

void f(vector<int>& v, list<char>& lst)
{
    v.rbegin()[3] = 7;           // 可以：随机访问迭代器
    lst.rbegin()[3] = '4';       // 错误：双向迭代器不支持 []
    *(+++++lst.rbegin()) = '4'; // ok!
}

```

此外，标准库还为**reverse_iterator**提供了 **==**、**!=**、**<**、**<=**、**>**、**>=**、**+** 和 **-**。

19.2.6 流迭代器

按照常规，I/O是通过流库（第21章）、图形用户界面系统（不在C++ 标准范围之内）或者C语言I/O函数（21.8节）完成的。这些I/O界面的基本目标在于读写各种类型的单个的值。标准库提供了4个迭代器类型，以使流I/O能够融入容器和算法的通用框架之中：

- **ostream_iterator**：用于向**ostream**写入（3.4节、21.2.1节）。
- **istream_iterator**：用于由**istream**读出（3.6节、21.3.1节）。
- **ostreambuf_iterator**：用于向流缓冲区写入（21.6.1节）。
- **istreambuf_iterator**：用于由流缓冲区读出（21.6.2节）。

其中的想法就是使输入和输出能以序列的方式呈现出来：

```

template <class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;

```



```

typedef basic_ostream<Ch, Tr> ostream_type;

ostream_iterator(ostream_type& s);
ostream_iterator(ostream_type& s, const Ch* delim); // 在每个输出值后写delim
ostream_iterator(const ostream_iterator&);
~ostream_iterator();

ostream_iterator& operator=(const T& val);           // 将val写到输出

ostream_iterator& operator*();
ostream_iterator& operator++();
ostream_iterator& operator++(int);
};

```

这个迭代器接受输出迭代器常规的写出和增量操作，并将它们转换为对`ostream`的 `<<` 操作。例如，

```

void f()
{
    ostream_iterator<int> os(cout); // 将int通过os写到cout
    *os = 7;                        // 输出7 (用cout << 7)
    ++os;                          // 准备好做下一次输出
    *os = 79;                      // 输出79
}

```

这里的 `++` 操作可能是去触发实际的输出操作，或者也可能什么都不做。不同实现可以采取不同的实现策略。因此，为了使代码可移植，在对`ostream_iterator`的任意两次赋值之间，都应该写一个 `++`。很自然，每个标准算法都是这样写的——否则它就无法对`vector`使用了。这也是如此定义`ostream_iterator`的原因。

`ostream_iterator`的实现很简单，留做练习（19.6[4]）。标准I/O支持多种不同的字符类型，`char_traits`（20.2节）描述字符类型在这方面的情况，它对于I/O和`string`都很重要。

针对`istream`的输入迭代器的定义与此类似：

```

template <class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag, T, Dist, const T*, const T*> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_istream<Ch, Tr> istream_type;

    istream_iterator(); // 输入结束
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator&);
    ~istream_iterator();

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator& operator++();
    istream_iterator operator++(int);
};

```

这个迭代器的描述目的就是使对容器的常规使用能触发从`istream`的 `>>` 操作。例如，

```

void f()
{
    istream_iterator<int> is(cin); // 通过is从cin读int
    int i1 = *is;                 // 读入一个int (用cin >> i1)
    ++is;                         // 准备好下一次读
}

```

```
int i2 = *is;      // 读入一个int
}
```

默认的*istream_iterator*表示输入结束，所以我们可以表述一个输入序列为：

```
void f(vector<int>& v)
{
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
}
```

为使这种程序能工作，标准库也为*istream_iterator*提供了 == 和 != 操作。

*istream_iterator*的实现不像*ostream_iterator*的实现那么简单，但它也还是比较简单的。*istream_iterator*的实现也留做练习（19.6[5]）。

19.2.6.1 流缓冲区

如21.6节的介绍，流I/O的基础想法就是，*ostream*和*istream*填充或者用掉缓冲区的内容，低级物理I/O在缓冲区之下做取来或者送走的工作。也可以跨过标准*iostream*的格式化工作，直接去与流缓冲区（21.6.4节）打交道。通过*istreambuf_iterator*和*ostreambuf_iterator*的概念，标准库为算法提供了这种能力：

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator
    : public iterator<input_iterator_tag, Ch, typename Tr::off_type, Ch*, Ch&> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;
    typedef basic_streambuf<Ch, Tr> streambuf_type;
    typedef basic_istream<Ch, Tr> istream_type;

    class proxy;                                // 协助类型

    istreambuf_iterator() throw();               // 缓冲区结束
    istreambuf_iterator(istream_type& is) throw(); // 从is的streambuf读入
    istreambuf_iterator(streambuf_type*) throw();
    istreambuf_iterator(const proxy& p) throw(); // 从p的streambuf读入

    Ch operator*() const;
    istreambuf_iterator& operator++();           // 前缀
    proxy operator++(int);                       // 后缀

    bool equal(istreambuf_iterator&);           // 两个streambuf都处于或都不处于eof
};
```

此外还提供了 == 和 !=。

从*streambuf*读入是比从*istream*读入更低级的操作，因此，*istreambuf_iterator*的界面也比*istream_iterator*的界面难看一些。但无论如何，只要*istreambuf_iterator*经过了正确初始化，按照正规方式使用的 *、++ 和 = 仍将具有它们通常的意义。

*proxy*类型是一个由实现定义的协助类型，用于使后缀的 ++ 操作不会给*streambuf*的实现强加上任何约束。在迭代器增量之后，*proxy*仍保存着结果值：

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch, Tr>::proxy {
    Ch val;
    basic_streambuf<Ch, Tr>* buf;

    proxy(Ch v, basic_streambuf<Ch, Tr>* b) : val(v), buf(b) {}
};
```

```
public:
    Ch operator*() { return val; }
};
```

`ostreambuf_iterator`的定义与此类似:

```
template <class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_streambuf<Ch, Tr> streambuf_type;
    typedef basic_ostream<Ch, Tr> ostream_type;

    ostreambuf_iterator(ostream_type& os) throw(); // 写入os的streambuf
    ostreambuf_iterator(streambuf_type*) throw();
    ostreambuf_iterator& operator=(Ch);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool failed() const throw(); // 如果遇到Tr::eof()则真
};
```

19.3 带检查迭代器

除了标准库所提供的迭代器外,程序员也可以提供自己的迭代器。在提供了一类新容器之后常常需要这样做。此外,有时提供一种新的迭代器,也是支持对已有容器另一种用法的很好方式。作为实例,我在这里讨论一种对容器的访问范围进行检查的迭代器。

使用标准容器能减少显式存储管理的代码量。使用标准算法能减少直接对容器中的元素寻址的次数。与传统的C风格代码相比,使用标准库并与语言中维护类型安全性的机制相结合,能够显著地减少运行时错误。然而,标准库还是要依靠程序员去避免超出容器范围的访问。如果由于某些偶然原因,对某个容器`x`访问了元素`x[x.size()+7]`,那么就会发生无法预期的事情——通常总是很坏的事。采用检查范围的`vector`(如`Vec`(3.7.2节))能对某些情况有所帮助。检查经由迭代器的每一次访问则可以处理更多的情况。

为使检查能做到这种程度,而又不给程序员强加严重的描述形式负担,我们就需要一种带检查迭代器,以及一种将它们附着到容器上的方便方式。为了做出`Checked_iter`,我们需要一个容器和一个到容器的迭代器。与约束器(18.4.4.1节)和插入器(19.2.4节)一样,我为创建`Checked_iter`提供了几个函数:

```
template<class Cont, class Iter> Checked_iter<Cont, Iter> make_checked(Cont& c, Iter i)
{
    return Checked_iter<Cont, Iter>(c, i);
}

template<class Cont> Checked_iter<Cont, typename Cont::iterator> make_checked(Cont& c)
{
    return Checked_iter<Cont, typename Cont::iterator>(c, c.begin());
}
```

这些函数提供的是一种记法上的便利,它们由参数推断出有关的类型,这样就不必显式地描述这些类型了。例如,

```

void f(vector<int>& v, const vector<int>& vc)
{
    typedef Checked_iter<vector<int>, vector<int>::iterator> CI;
    CI p1 = make_checked(v, v.begin()+3);
    CI p2 = make_checked(v);           // 默认约定: 指向第一个元素

    typedef Checked_iter<const vector<int>, vector<int>::const_iterator> CIC;
    CIC p3 = make_checked(vc, vc.begin()+3);
    CIC p4 = make_checked(vc);

    const vector<int>& vv = v;
    CIC p5 = make_checked(v, vv.begin());
}

```

默认情况下, `const`容器具有`const`迭代器, 因此它们的`Checked_iter`也必须是`const`迭代器。迭代器`p5`也显示了一种从非`const`容器创建`const`迭代器的方法。

上面例子也说明了为什么`Checked_iter`需要两个模板参数: 其中的一个是容器类型, 另一个表示`const`或者非`const`迭代器。

这些`Checked_iter`的名字会变得相当长而笨拙, 但在将迭代器用做通用型算法的参数时, 这一情况毫无影响。例如,

```

template<class Iter> void mysort(Iter first, Iter last);

void f(vector<int>& c)
{
    try {
        mysort(make_checked(c), make_checked(c, c.end()));
    }
    catch (out_of_bounds) {
        cerr<<"oops: bug in mysort()\n";
        abort();
    }
}

```

在讨论这个算法的早期版本的地方, 正好是我对范围错误问题提出了许多质疑, 说明采用带检查的迭代器很有意义的地方。

`Checked_iter`的表示就是一个指向容器的指针, 再加一个指到容器里面的迭代器:

```

template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    Iter curr; // 当前位置迭代器
    Cont* c;   // 当前容器指针

    // ...
};

```

从`iterator_traits`派生是定义所需要的`typedef`的一种技术。另一种明显方式, 是从`iterator`派生, 在当前情况下用起来比较啰嗦 (就像用于`reverse_iterator`一样; 19.2.5节)。如同我们并不要求迭代器一定是类一样, 也没有要求说迭代器类必须从`iterator`派生。

`Checked_iter`操作都相当简单:

```

template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    // ...
public:
    void valid(Iter p) const

```

```

{
    if (c->end() == p) return;
    for (Iter pp = c->begin(); pp != c->end(); ++pp) if (pp == p) return;
    throw out_of_bounds();
}

friend bool operator==(const Checked_iter& i, const Checked_iter& j)
{
    return i.c == j.c && i.curr == j.curr;
}

// 无默认初始式
// 用默认的复制构造函数和复制赋值

Checked_iter(Cont& x, Iter p) : c(&x), curr(p) { valid(p); }
Checked_iter(Cont& x) : c(&x), curr(x.begin()) { }

reference operator*() const
{
    if (curr == c->end()) throw out_of_bounds();
    return *curr;
}

pointer operator->() const
{
    if (curr == c->end()) throw out_of_bounds();
    return &*curr;
}

Checked_iter operator+(difference_type d) const // 只对随机访问迭代器
{
    if (c->end() - curr < d || d < curr - c->begin()) throw out_of_bounds();
    return Checked_iter(c, curr + d);
}

reference operator[](difference_type d) const // 只对随机访问迭代器
{
    if (c->end() - curr <= d || d < curr - c->begin()) throw out_of_bounds();
    return curr[d];
}

Checked_iter& operator++() // 前缀 ++
{
    if (curr == c->end()) throw out_of_bounds();
    ++curr;
    return *this;
}

Checked_iter operator++(int) // 后缀 ++
{
    Checked_iter tmp = *this;
    ++*this; // 通过前缀 ++ 检查
    return tmp;
}

Checked_iter& operator--() // 前缀 --
{
    if (curr == c->begin()) throw out_of_bounds();
    --curr;
    return *this;
}

```

```

Checked_iter operator--(int) // 后缀 --
{
    Checked_iter tmp = *this;
    --*this; // 通过前缀 -- 检查
    return tmp;
}

difference_type index() const { return curr - c.begin(); } // 只对随机访问迭代器

Iter unchecked() const { return curr; }

// +, -, < 等 (19.6[6])
};

```

这种 **Checked_iter** 只能用一个指向特定容器的特定迭代器进行初始化。在一个更完善的实现里，应该为随机访问迭代器提供一个更有效的 **valid()** 版本 (19.6[6])。当一个 **Checked_iter** 初始化之后，任何改变它位置的操作都将被检查，以保证这个迭代器始终指在对应容器的内部。企图将迭代器移出容器的操作将导致抛出 **out_of_bounds** 异常。例如，

```

void f(list<string>& ls)
{
    int count = 0;
    try {
        Checked_iter<list<string>> p(ls, ls.begin());
        while (true) {
            ++p; // 早晚将达到末端
            ++count;
        }
    }
    catch(out_of_bounds) {
        cout << "overrun after " << count << " tries\n";
    }
}

```

Checked_iter 知道它所指的是哪一个容器，这就使它能够捕捉到一些（并不是全部）情况，其中由于容器操作使指向这个容器的迭代器变成非法的了 (16.3.6 节、16.3.8 节)。要想抵御所有这类情况，将需要另外一种不同的而且代价更高昂的迭代器 (19.6[7])。

注意，后增量（后缀 ++）涉及到一个临时量，前增量则不需要。由于这个原因，对迭代器最好是用 **++p** 而不是 **p++**。

由于 **Checked_iter** 保存着一个指向容器的指针，因此它不能直接用于内部数组。如果真有这种需要，可以用 **c_array** (17.5.4 节)。

为了使带检查迭代器的概念更完整，我们必须使它们易于使用。存在着两条基本途径：

- [1] 定义一种带检查容器类型，其行为就像其他的容器，但它只提供了更有限的一组构造函数和 **begin()**、**end()** 等，支持 **Checked_iter** 而不是常规迭代器。
- [2] 定义一种句柄，它可以用任意容器进行初始化，并为其容器提供一些带检查的访问函数 (19.6[8])。

下面的模板将带检查迭代器附着到容器上：

```

template<class C> class Checked : public C {
public:
    explicit Checked(size_t n) : C(n) {}
    Checked() : C() {}

```

```

typedef Checked_iter<C> iterator;
typedef Checked_iter<C, C::const_iterator> const_iterator;

iterator begin() { return iterator(*this, C::begin()); }
iterator end() { return iterator(*this, C::end()); }
const_iterator begin() const { return const_iterator(*this, C::begin()); }
const_iterator end() const { return const_iterator(*this, C::end()); }

typename C::reference_type operator[] (typename C::size_type n)
{ return Checked_iter<C>(*this)[n]; }

C& base() { return *this; }    // 取得基础容器
};

```

这就使我们能够写：

```

Checked< vector<int> > vec(10);
Checked< list<double> > lst;

void f()
{
    int i1 = vec[5];           // ok
    int i2 = vec[15];          // 抛出out_of_bounds
    // ...
    mysort(vec.begin(), vec.end());
    copy(vec.begin(), vec.end(), lst.begin());
}

```

提供明显多余的`base()`函数，就是为了使`Checked()`的界面更像容器句柄的界面。容器的句柄通常并不提供到它们的容器的隐式转换。

如果一个容器改变了大小，所有指向它里面的迭代器（包括`Checked_iter`）就可能变得非法了。在这种情况下，`Checked_iter`可以重新初始化：

```

void g(vector<int>& vi)
{
    Checked_iter< vector<int> > p(vi);
    // ..
    int i = p.index();           // 取得当前位置
    vi.resize(100);              // p变为非法
    p = Checked_iter< vector<int> >(vi, vi.begin()+i); // 恢复当前位置
}

```

那个老的非法的当前位置丢失了。我提供了`index()`，作为一种提取下标以便能够恢复`Checked_iter`的方式。

19.3.1 异常、容器和算法

你可能会争辩说，同时使用标准库和带检查迭代器就像是同时系上了安全带和保险吊索：任何一个就已经能保安全了。然而，经验说明，对于许多人和许多应用而言，有点偏执还是很合理的——特别是在程序正历经频繁修改，其中涉及到许多人的时期。

使用运行时检查的一种方式是在排除程序错误期间让它们留在代码中，在程序发布之前就将其过滤掉。这种实践方式被人比喻为在靠近海边踱步时穿着救生衣，在出发进入大海之前却将它抛弃了。当然，有些运行时检查的使用确实会带来显著的额外时间和空间开销，这样，强求在所有时间都做这种检查也不太实际。在任何情况下，做优化时不进行实测都是不明智的，

所以，在清除这些检查之前，应该先做试验，看看清除它们之后是不是真能取得很有价值的性能改善。在做这种试验时，我们必须能很方便地去掉这些运行时检查（见24.3.7.1节）。一旦完成实测，我们就可能从运行时最关键的一些代码中去掉运行时检查功能（并期望它们已经通过了最彻底的测试），并保留其余代码中的检查，作为一种相对便宜的保险措施。

采用**Checked_iter**使我们能够检查出许多错误，当然，它也不可能使我们更容易从这种错误中恢复执行。人们很少写出预防每一个++、--、*、[]、->和=可能抛出的异常的100%健壮的代码。这就使我们可能采取两种比较明显的策略：

- [1] 在最接近异常抛出点的地方捕捉它，使写异常处理器的人能有相当多的机会了解究竟是什么出了问题，并采取适当的应对动作。
- [2] 在程序的高层次上捕捉异常，抛弃计算中很大的一个部分，并考虑在失败的可疑计算过程中写出的所有数据结构（或许根本没有这种数据结构，或许可以合情合理地去检查它们）。

在捕捉到一个来自程序中未知部分的异常后，假设非期望状态中不存在任何数据结构并继续前进，这些做法都是极不负责任的，除非还有进一步的错误处理层次能捕捉随后出现的错误。这方面的一个简单事例就是在接受程序结果之前（由人或者计算机）所做的最后检查。在这些情况下，与试图在某个低层次捕捉每一个异常相比，愉快地继续下去是更简单也更廉价的。这也是多层错误恢复模式（14.9节）可能导致简化的一个具体实例。

19.4 分配器

allocator（分配器）被用于将算法和容器的实现（它们都需要分配存储）隔离于物理存储的细节之外。一个分配器提供了一套分配与释放存储的标准方式，以及一套用做指针类型和引用类型的标准名字。与迭代器一样，分配器也是一种纯粹的抽象，任何在行为上像分配器的类型都是分配器。

标准库提供了一个标准分配器，其目的就是能为给定实现的大部分用户提供很好的服务。除此之外，用户也可以提供代表了对存储的其他观点的分配器。例如，我们可以写一些分配器去使用共享存储、带废料搜集的存储、来自预分配对象池的存储（19.4.2节）等。

标准库容器和算法都通过某个分配器提供的功能获得和访问存储。这样，通过提供一个新的分配器，我们也就为标准容器提供了一种新的不同的存储使用方式。

19.4.1 标准分配器

在<memory>里的标准**allocator**模板用**operator new()**（6.2.6节）分配存储，所有的标准容器在默认方式下都使用它：

```
template <class T> class std::allocator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef T* pointer;
    typedef const T* const_pointer;

    typedef T& reference;
    typedef const T& const_reference;
```



```

pointer address(reference r) const { return &r; }
const_pointer address(const_reference r) const { return &r; }

allocator() throw();
template <class U> allocator(const allocator<U>&) throw();
~allocator() throw();

pointer allocate(size_type n, allocator<void>::const_pointer hint = 0); // 给n个T的空间
void deallocate(pointer p, size_type n); // 释放n个T, 不销毁

void construct(pointer p, const T& val) { new(p) T(val); } // 用val初始化 *p
void destroy(pointer p) { p->~T(); } // 销毁 *p, 不释放

size_type max_size() const throw();

template <class U>
struct rebind { typedef allocator<U> other; }; // 效果: typedef allocator<U> other
};

template <class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template <class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();

```

allocate(n) 操作为n个对象分配空间, 这块空间可以由对应的**deallocate(p, n)** 释放。注意, **deallocate()** 也以元素的个数n为参数, 这样就可能在只维持最少量的有关被分配存储的信息的条件下, 尽可能地优化分配器。在另一方面, 这种分配器也要求用户在调用**deallocate()** 时总能提供正确的n值。注意, **deallocate()** 与运算符**delete()** (6.2.6节) 的不同之处在于它的参数必须非0。

默认**allocator**用**operator new(size_t)** 获取存储, 用**operator delete(void*)** 操作释放它。这也意味着在存储耗尽的情况下可能调用**new_handler()**, 可能抛出**std::bad_alloc**异常 (6.2.6.2节)。

请注意, **allocate()** 不必每次都去调用低级分配系统。通常, 一个更好的策略是让分配器管理一个关于空间的自由表, 以便能在最小的时间开销下提交存储 (19.4.2节)。

allocate() 的可选参数**hint**完全依赖于实现。不过它的意图就是在那些局部性特别重要的系统中为分配器提供帮助。例如, 在分页系统中, 一个分配器应尽可能试图将相关对象的空间分配在同一个页面里。在下面极度简化后的专门化中, **hint**参数的类型是**pointer**:

```

template <> class allocator<void> {
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // 注意: 不是reference
    typedef void value_type;
    template <class U>
    struct rebind { typedef allocator<U> other; }; // 效果: typedef allocator<U> other
};

```

allocator<void>::pointer类型被当做一个通用指针类型, 在所有的标准分配器里都用**void***。

除非在有关分配器的文档里有另外的说法, 否则, 用户在调用**allocate()** 时都存在两种合理选择:

[1] 不给提示。

[2] 用一个指向对象的指针作为提示, 该对象常常与新对象一起使用。例如, 同属一个序列的前一个元素。

采用分配器的目的就是使容器的实现者能不必直接与原始存储打交道。作为实例，下面考虑一个**vector**实现可能如何使用存储：

```
template <class T, class A = allocator<T> > class vector {
public:
    typedef typename A::pointer iterator;
    // ...
private:
    A alloc;           // 分配器对象
    iterator v;         // 指向元素的指针
    // ...
public:
    explicit vector(size_type n, const T& val = T(), const A& a = A())
        : alloc(a)
    {
        v = alloc.allocate(n);
        for(iterator p = v; p < v+n; ++p) alloc.construct(p, val);
        // ...
    }

    void reserve(size_type n)
    {
        if (n <= capacity()) return;

        iterator p = alloc.allocate(n);
        iterator q = v;

        while (q < v+size()) {                // 复制现存元素
            alloc.construct(p++, *q);
            alloc.destroy(q++);
        }
        alloc.deallocate(v, capacity());      // 释放原有空间
        v = p-size();
        // ...
    }
    // ...
};
```

allocator的操作都是基于**pointer**和**reference**这两个**typedef**表述的，这就使用户有可能提供另外的替代类型去访问存储。一般而言，做这件事情是非常困难的。比如说，在C++语言里就不可能定义出一个完美的引用类型。当然，语言和库的实现者可以利用这些**typedef**去支持某些常规用户无法提供的类型。一个例子是可以有一个分配器，提供对持续性存储器的访问。另一个例子可以是一种“长”指针类型，用于访问超出常规指针（通常是32位）寻址范围的主存。

常规用户可以为分配器提供一种不寻常的指针类型，以服务于特殊的用途。对引用就无法做与此等价的事情。对于试验性系统或者专用系统，这种约束还可以接受。

分配器的设计使它很容易处理通过其模板参数描述的类型对象。当然，大部分容器的实现都需要另一些类型的对象。例如，**list**的实现将需要分配**Link**对象，通常这种**Link**对象必须用它们所在的**list**的分配器进行分配的。

提供那个古怪的**rebind**类型，就是为使一个分配器能够分配任意类型的对象。考虑

```
typedef typename A::template rebind<Link>::other Link_alloc; // “模板”，见C.13.6节
```

如果**A**是一个**allocator**，那么**rebind<Link>::other**就被**typedef**定义为代表**allocator<Link>**，所

以，上面这个**typedef**也就是下面描述的一种间接表述方式：

```
typedef allocator<Link> Link_alloc;
```

这种间接表述使我们不必直接提及**allocator**，它借助于模板参数**A**去描述**Link_alloc**类型。例如，

```
template <class T, class A = allocator<T> > class list {
private:
    class Link { /* ... */ ;
        typedef typename A::rebind<Link>::other Link_alloc;    // <Link>分配器
        Link_alloc a;    // link分配器
        A alloc;        // list分配器
        // ...
public:
    typedef typename A::pointer iterator;
    // ...
    iterator insert(iterator pos, const T& x)
    {
        Link_alloc::pointer p = a.allocate(1);    // 取得一个Link
        // ...
    }
    // ...
};
```

Link是**list**的一个成员，所以它也用一个分配器参数化（13.2.1节）。由此可知，带有不同分配器的出自**list**的**Link**就属于不同的类型，就像**list**本身一样（17.3.3节）。

19.4.2 一个用户定义分配器

容器的实现常常需要一次一个地**allocate()**或者**deallocate()**对象。对于**allocate()**的朴素实现，这就意味着对运算符**new**的大量调用，在这样使用的情况下，运算符**new**的有些实现效率并不高。作为用户定义分配器的一个例子，我将展示一种模式，采用一些固定大小存储块的存储池，以使分配器的分配效率比常规的更通用的**operator new**高一些。

我正好有一个池分配器，它所做的基本上就是正确的事情，但它的界面不正确（因为它是一年以前设计的，是在分配器被发明之前）。这个**Pool**类实现了一种固定大小的元素池的概念，用户可以在这里做快速的分配和释放。这是一个低级类型，它直接操纵存储并考虑了对齐问题：

```
class Pool {
    struct Link { Link* next; };
    struct Chunk {
        enum { size = 8*1024-16 };    // 比8K略小，使一个块可以放入8K中
        char mem[size];              // 分配区放在前面，以取得严格的对齐
        Chunk* next;
    };
    Chunk* chunks;
    const unsigned int esize;
    Link* head;
    Pool(Pool&);                      // 防止复制
    void operator=(Pool&);           // 防止复制
    void grow();                      // 扩大存储池
};
```

```

public:
    Pool(unsigned int n);    // n是元素大小
    ~Pool();

    void* alloc();           // 分配一个元素
    void free(void* b);      // 将一个元素放回池中
};

inline void* Pool::alloc()
{
    if (head==0) grow();
    Link* p = head;         // 返回第一个元素
    head = p->next;
    return p;
}

inline void Pool::free(void* b)
{
    Link* p = static_cast<Link*>(b);
    p->next = head;         // 将b放回作为第一个元素
    head = p;
}

Pool::Pool(unsigned int sz)
    : esize(sz<sizeof(Link)?sizeof(Link):sz)
{
    head = 0;
    chunks = 0;
}

Pool::~~Pool() // 释放所有的块
{
    Chunk* n = chunks;
    while (n) {
        Chunk* p = n;
        n = n->next;
        delete p;
    }
}

void Pool::grow() // 分配新块，将它组织成大小为esize的链表
{
    Chunk* n = new Chunk;
    n->next = chunks;
    chunks = n;

    const int nelem = Chunk::size/esize;
    char* start = n->mem;
    char* last = &start[(nelem-1)*esize];
    for (char* p = start; p<last; p+=esize)
        reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
    reinterpret_cast<Link*>(last)->next = 0;
    head = reinterpret_cast<Link*>(start);
}

```

为增加一些真实性，我把不加改变的**Pool**作为我的分配器实现的一部分，而不打算重写它去得到正确的界面。这个池分配器就是想做单一元素的快速分配和释放，这也就是我的**Pool**所支持的方式。将这个实现扩充到能处理任意数目的对象和任意大小的对象（如**rebind**所要求的）也留做练习（19.6[9]）。

有了**Pool**之后，**Pool_alloc**的定义十分简单：

```
template <class T> class Pool_alloc {
private:
    static Pool mem;    // sizeof(T) 的元素的池
public:
    // 像标准分配器 (19.4.1节)
};

template <class T> Pool Pool_alloc<T>::mem(sizeof(T));

template <class T> Pool_alloc<T>::Pool_alloc() {}

template <class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
    if (n == 1) return static_cast<T*>(mem.alloc());
    // ...
}

template <class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
    if (n == 1) {
        mem.free(p);
        return;
    }
    // ...
}
```

现在，这个分配器就可以按显然的方式使用了：

```
vector<int, Pool_alloc<int>> v;
map<string, number, Pool_alloc<pair<const string, number>>> m;

// 使用方式完全不变

vector<int> v2 = v;    // 错误：不同的分配器参数
```

我选择将**Pool_alloc**里的**Pool**作为静态的，因为标准库对于标准容器所用的分配器有一个限制：允许标准容器的实现将它的分配器类型的每个对象看成是等价的。这将带来明显的性能优势。例如，由于有这个限制，对于**Link**对象里的分配器的存储就无须分开另外安排（这种分配器通常是由这些**Link**所在的那个容器的分配器参数化而来的；19.4.1节），而且，那些访问两个序列的操作（如**swap()**）也不需要检查被处理的对象是否具有同样的分配器。当然，这一限制并不意味着这种分配器不能采用在每个对象里放数据的方式。^①

在应用这类优化之前，应当弄清确实需要它。我期望许多默认的**allocator**都能实现这类经典的C++优化，从而也省去你的忧虑。

19.4.3 广义的分配器

allocator是有关通过模板参数向容器传递信息的思想（13.4.1节、16.2.3节）的一种简化和优化后的变形。例如，要求容器中的每个元素都通过容器的分配器进行分配也很有意义。当然，如果允许两个同样类型的**list**采用不同的分配器，那么**splice()**（17.2.2.1节）就无法通

① 这里提出的问题是，一个分配器是否在其所分配的每个对象里放入某种信息，以便用于自己的工作（例如释放存储），或者其他操作的实现（例如用于确定某些操作的合法性或操作方式）。——译者注

过重新链接的方式实现了。这时只能换一种方式，将`splice()`定义成基于元素复制，以防出现偶发的情况，其中我们可能想将采用一种分配器的`list`元素粘接到采用同样分配器类型的另一分配器的另一个`list`去。与此类似，如果允许分配器完全通用，那么，使一个分配器能为任意类型分配元素的`rebind`机制将不得不做得更精细。由此，标准分配器被假定为不在每个对象里存放数据，而且标准的实现可以利用这一情况。

令人吃惊的是，反对分配器里在对象内存放信息这种貌似极严格的限制实际并不很严重。大部分分配器根本不需要在对象内存放数据，没有这种数据可以运行得更快。分配器还是可以保存着有关分配器类型的数据。如果需要不同的数据，就可以使用不同的分配器类型。例如，

```
template<class T, class D> class My_alloc { // 用D为T实现的分配器
    D d; // 为My_alloc<T, D>所需的数据
    // ...
};

typedef My_alloc<int, Persistent_info> Persistent;
typedef My_alloc<int, Shared_info> Shared;
typedef My_alloc<int, Default_info> Default;

list<int, Persistent> lst1;
list<int, Shared> lst2;
list<int, Default> lst3;
```

这里的表`lst1`、`lst2`和`lst3`具有不同类型。因此，在操作两个这样的表时，我们就必须使用通用的算法，而不能用特殊的表操作（17.2.2.1节）。这意味着所做的是复制而不是重新链接，这时具有不同分配器就不会引起问题。

对于分配器中不允许在每个对象里放数据的限制是强制性的，这是因为标准库在运行时间和空间效率上有严格要求。例如，对于一个表，由于分配器的数据造成的额外空间开销不会有显著影响，但如果表里的每个链接都付出这种开销，事情就会很严重了。

现在考虑，当标准库在效率方面的限制并不重要的时候，我们还可以采用哪些分配器技术。这也就是在一些非标准库的情况，或者是某些服务于特殊目的的标准库实现的情况，它们可能从根本上就不打算为程序里的每个数据结构和每个类型提供最高的性能。在这些情况下，分配器可以被用来携带某一类信息，这些信息常常是从某个公共基类继承来的（16.2.2节）。举例说，有可能将一个分配器设计为能回答对象被分配在哪里的询问，可以给出说明对象布局的数据，可以回答例如“这个元素在这个容器里吗”之类的问题等。它也可以为容器提供一种控制，使容器的行为就像是一个为持续性存储器工作的高速缓存存储器，或者提供容器与其他对象之间的关联，等等。

按照这种方式，任何服务都可能以对常规容器操作透明的形式提供。然而，最好还是将有关数据存储的问题和数据使用的问题分开，后一件事情并不属于某个广义的分配器，但它也可能通过单独的模板参数提供。

19.4.4 未初始化的存储

除了标准`allocator`之外，`<memory>`头文件还为处理未初始化的存储提供了几个函数。这些函数都是很危险的，但有时又是必不可少的功能，因为这时我们需要用类型名`T`去引用一块足够保存一个`T`类型的对象的空间，而不是引用一个构造完好的`T`类型的对象。

标准库为把一些值复制到未初始化的存储块提供了三种方式：

```

template <class In, class For>
For uninitialized_copy(In first, In last, For res)    // 复制到res
{
    typedef typename iterator_traits<For>::value_type V;

    while (first != last)
        new (static_cast<void*>(&*res++)) V(*first++);    // 在res构造
    return res;
}

template <class For, class T>
void uninitialized_fill(For first, For last, const T& val)    // 复制到 [first, last)
{
    typedef typename iterator_traits<For>::value_type V;

    while (first != last) new (static_cast<void*>(&*first++)) V(val);    // 在first构造
}

template <class For, class Size, class T>
void uninitialized_fill_n(For first, Size n, const T& val)    // 复制到 [first, first + n)
{
    typedef typename iterator_traits<For>::value_type V;

    while (n-->0) new (static_cast<void*>(&*first++)) V(val);    // 在first构造
}

```

这些函数从本质上就是想提供给容器和算法的实现者的。例如，利用这些函数很容易实现 `reverse()` 和 `resize()` (16.3.8节、19.6[10])。事情很清楚，如果某个未初始化的对象从容器内部逃出，到了普通用户手里，那将会是极大的不幸。另见E.4.4节。

算法为达到可接受的性能常需要一些临时空间。这种临时空间最好能通过一个操作分配，但直到实际需要用其中的位置时再做初始化。为此，标准库提供了一对函数来分配和释放未初始化的空间：

```

template <class T> pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t); // 分配，不初始化
template <class T> void return_temporary_buffer(T*);    // 释放；不销毁

```

`get_temporary_buffer<X>(n)` 操作设法去分配一些空间，其中足以存放 n 个或者更多个类型 X 的对象。如果成功分配到这样的存储，它就返回指向未初始化的空间开始处的指针，以及能放入这块空间里的类型 X 对象的数目；否则，返回对偶值中的 `second` 就是 0。这里的想法是，系统可能维持着一些固定大小的缓冲区专供快速分配之用，所以，要求 n 个对象的空间也可能得到的多于 n 个，也可能不足 n 个。由于这种情况，使用 `get_temporary_buffer()` 的一种方式是先按照最优情况去多申请，而后去使用当时能够分到的那些空间。

由 `get_temporary_buffer()` 得到的缓冲区必须通过 `return_temporary_buffer()` 释放，以便其他地方再用。与 `get_temporary_buffer()` 只分配不构造相对应，`return_temporary_buffer()` 也是只释放不销毁。因为 `get_temporary_buffer()` 是低级机制，而且很可能为管理临时性缓冲区专门做了优化，因此，不应该将它用做 `new` 或者 `allocator::allocate()` 的替代品去获取长期使用的存储区。

向一个序列里写的标准算法都假定该序列的元素已经做了初始化。也就是说，这些算法在完成写动作时用的是赋值而不是复制构造函数。这样，我们就不能用未初始化的存储作为算法的直接目标。这种情况可能很不幸，因为赋值可能比初始化的代价高得多。此外，我们也根本不会关心准备复写掉的那些值（要不然为什么要复写掉它们）。解决的办法是使用

<memory> 里的 `raw_storage_iterator`，它做的是初始化而不是赋值：

```
template <class Out, class T>
class raw_storage_iterator : public iterator<output_iterator_tag, void, void, void, void> {
    Out p;
public:
    explicit raw_storage_iterator(Out pp) : p(pp) {}
    raw_storage_iterator& operator*() { return *this; }
    raw_storage_iterator& operator=(const T& val) {
        T* pp = &*p;
        new(pp) T(val);    // 将val放到pp (10.4.11节)
        return *this;
    }
    raw_storage_iterator& operator++() { ++p; return *this; }
    raw_storage_iterator operator++(int) {
        raw_storage_iterator t = *this;
        ++p;
        return t;
    }
};
```

例如，我们可以写出如下模板，它将 `vector` 的内容复制到一个缓冲区：

```
template<class T, class A> T* temporary_dup(vector<T, A>& v)
{
    pair<T*, ptrdiff_t> p = get_temporary_buffer<T>(v.size());
    if (p.second < v.size()) {    // 检查是否有足够的存储可用
        if (p.first != 0) return temporary_buffer(p.first);
        return 0;
    }
    copy(v.begin(), v.end(), raw_storage_iterator<T*, T>(p.first));
    return p.first;
}
```

如果用的是 `new` 而不是 `get_temporary_buffer()`，那么初始化早就已经做过了。如果避免了初始化，那么必须用 `raw_storage_iterator` 去处理这种未初始化的存储。在上面这个例子里，`temporary_dup()` 的调用者应负责对它所取得的指针调用 `return_temporary_buffer()`。

19.4.5 动态存储

用于实现 `new` 和 `delete` 运算符的功能在 <new> 里声明：

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t {};
extern const nothrow_t nothrow;    // 分配时不抛出异常的指示符

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete(void*, const nothrow_t&) throw();

void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();

void* operator new[](size_t, const nothrow_t&) throw();
```



```

void operator delete[] (void*, const nothrow_1&) throw();

void* operator new (size_t, void* p) throw() { return p; } // 放置 (10.4.11节)
void operator delete (void* p, void*) throw() { } // 什么也不做

void* operator new[] (size_t, void* p) throw() { return p; }
void operator delete[] (void* p, void*) throw() { } // 什么也不做

```

带有空异常描述 (*exception-specification*) (14.6节) 的 `operator new()` 或 `operator new[]()` 不能通过抛出 `std::bad_alloc` 异常发出存储耗尽的信号, 在分配失败时它们返回0。`new` 表达式 (`new_expression`) (6.2.6.2节) 将检测由带有空异常描述的分配函数返回的值, 如果返回值是0, 就不调用构造函数并返回值0。特别地, 那些 `nothrow` 分配器将用返回0指明分配失败, 而不是抛出 `bad_alloc`。例如,

```

void f()
{
    int* p = new int[100000]; // 可能抛出 bad_alloc

    if (int* q = new(nothrow) int[100000]) { // 不会抛出异常
        // 分配成功
    }
    else {
        // 分配失败
    }
}

```

这就使我们可以为存储分配采用某种在异常发生之前处理错误的策略。

19.4.6 C风格的分配

C++ 从C继承了一个用于动态存储的函数界面, 它们都能在 `<cstdlib>` 里找到:

```

void* malloc (size_t s); // 分配s个字节
void* calloc (size_t n, size_t s); // 分配n乘s个字节, 初始化为0
void free (void* p); // 释放由malloc()或calloc()分配的空间
void* realloc (void* p, size_t s); // 将p所指数组的大小变为s,
// 如做不到就分配s个字节,
// 将p所指数组复制过去并释放p

```

应该避免使用这些函数, 尽量用 `new`、`delete` 和标准容器。这些函数处理的都是未初始化的存储。特别是 `free()` 将不会对它所释放的存储调用析构函数。实现 `new` 和 `delete` 时有可能使用了这些函数, 但并不保证这样做。举例来说, 用 `new` 分配一个对象而后用 `free()` 删除, 那就是自找麻烦。如果你感到需要用 `realloc()`, 请换个想法, 考虑依赖于标准容器, 那样做通常会更简单也更有效 (16.3.5节)。

标准库还提供了一组函数, 其目的都在于高效地完成字节操作。因为C原本是通过 `char*` 指针访问无类型的字节, 所以这些函数都在 `<cstring>` 里。所有 `void*` 指针在这些函数里面当做 `char*` 指针一样处理:

```

void* memcpy (void* p, const void* q, size_t n); // 复制不重叠的区域
void* memmove (void* p, const void* q, size_t n); // 复制可能重叠的区域

```

与 `strcpy()` (20.4.1节) 一样, 这些函数从 `q` 向 `p` 复制 `n` 个字节并返回 `p`。用 `memmove()` 复制的区域可以有重叠, 而 `memcpy()` 假定其区域不重叠, 一般它也利用这个假定被优化。类似地:

```
void* memchr(const void* p, int b, size_t n); // 像strchr() (20.4.1节): 在p[0]..p[n-1]中找b
int memcmp(const void* p, const void* q, size_t n); // 像strcmp(): 比较字节序列
void* memset(void* p, int b, size_t n); // 将n个字节设置为b, 返回p
```

许多实现都为这些函数提供了高度优化的版本。

19.5 忠告

- [1] 在写一个算法时, 设法确定需要用哪种迭代器才能提供可接受的效率, 并(只)使用这种迭代器所支持的操作符去表述算法; 19.2.1节。
- [2] 当给定的迭代器参数提供了多于算法所需的最小支持时, 请通过重载为该算法提供效率更高的实现; 19.2.3节。
- [3] 利用*iterator_traits*为不同迭代器类别描述适当的算法; 19.2.2节。
- [4] 记住在*istream_iterator*和*ostream_iterator*的访问之间使用 ++; 19.2.6节。
- [5] 用插入器避免容器溢出; 19.2.4节。
- [6] 在排错时使用额外的检查, 后面只在必须时才删除这些检查; 19.3.1节。
- [7] 多用 ++p, 少用 p++; 19.3节。
- [8] 使用未初始化的存储去改善那些扩展数据结构的算法的性能; 19.4.4节。
- [9] 使用临时缓冲区去改善需要临时数据结构的算法的性能; 19.4.4节。
- [10] 在写自己的分配器之前三思; 19.4节。
- [11] 避免*malloc()*、*free()*、*realloc()*等; 19.4.6节。
- [12] 你可以通过为*rebind*所用的技术去模拟对模板的*typedef*; 19.4.1节。

19.6 练习

1. (*1.5) 实现18.6.7节的*reverse()*。提示: 参见19.2.3节。
2. (*1.5) 写一个输出迭代器*Sink*, 它并不实际向任何地方写。*Sink*在什么情况下有用?
3. (*2) 实现*reverse_iterator* (19.2.5节)。
4. (*1.5) 实现*ostream_iterator* (19.2.6节)。
5. (*2) 实现*istream_iterator* (19.2.6节)。
6. (*2.5) 完成*Checked_iter* (19.3节)。
7. (*2.5) 重新设计*Checked_iter*, 让它检查非法的迭代器。
8. (*2) 设计并实现一个句柄类, 它的行为像是一个容器的代理, 并且为用户提供了一个完全的容器界面。它的实现应该包含一个指向容器的指针, 再加上对范围做检查的容器操作的实现。
9. (*2.5) 完成并重新设计*Pool_alloc* (19.4.2节), 使之能提供标准库*allocator* (19.4.1节)的所有功能。比较*allocator*和*Pool_alloc*的性能, 看看是否有理由在你的系统中使用*Pool_alloc*。
10. (*2.5) 实现*vector*, 采用分配器而不用*new*和*delete*。

第20章 串

宁要标准，
不要不落俗套。
——Shrunk & White

串——字符——*char_traits*——*basic_string*——迭代器——元素访问——构造函数
——错误处理——赋值——转换——比较——插入——拼接——查找和替换——大小和
容量——串I/O——C风格字符串——字符分类——C库函数——忠告——练习

20.1 引言

串是字符的序列。标准库*string*提供了处理串的各种操作，例如，下标（20.3.3节）、赋值（20.3.6节）、比较（20.3.8节）、附加（20.3.9节）、拼接（20.3.10节）和子串检索（20.3.11节）等。标准并没有提供通用的子串功能，所以这里将提供一个作为标准串使用的例子（20.3.11节）。一个标准串本质上可以是任何种类的字符的串（20.2节）。

经验表明，设计好一个完美的*string*是不可能的。人们对此的口味、期望和需要大相径庭，所以标准库的*string*并不很理想，要是让我做，或许会采取一些不同的设计决策，让你做也一样。但无论如何，这个*string*为许多需要提供了很好的服务，为满足进一步的需要而提供辅助函数也很容易，*std::string*已经得到广泛理解而且可以使用。在许多情况下，这些因素比我们能提供的小改进更重要一些。写出一个串类具有极大的教育价值（11.12节、13.2节），但对于那些意欲广泛使用的代码而言，还是应该用标准库的*string*。

C++ 从C继承来的字符串概念是以0结束的*char*数组，还有一组操作这样的C风格字符串的函数（20.4.1节）。

20.2 字符

“字符”本身就是个有趣的概念。请考虑字符C。你看到的C是纸面上（或屏幕上）的一段曲线，我在许多个月之前将它键入我的计算机。在那里，它在一个8位的字节里以数值67的形式生存。它是拉丁字母表的第三个字母，第6个原子（碳）的常用缩写形式，以及很偶然的，一种程序设计语言的名字（1.6节）。在有关字符串的程序设计的环境中有一个重要情况，在那些有着习惯意义的统称为字符的弯曲线与数值之间，存在着一种对应关系。情况还进一步复杂化，同一个字符在不同字符集里有着不同的数值，并不是每个字符集都有对应于每个字符的值，许多不同的字符集都在广泛使用着。一个字符集就是在字符（某些习惯性的符号）与整数值之间的一种映射。

C++ 程序员通常都假定能够用美国字符集（ASCII），但C++ 也允许在程序员的环境里缺少某些字符的可能性。例如，在缺少如[和{字符时，可以使用关键字或者二联符序列（C.3.1节）。

包含ASCII里所没有的字符的字符集提出了一个大挑战。许多语言（如中文、丹麦文、法文、冰岛文、日文）只用ASCII都无法正常地写出来。更糟糕的是，用于这些语言的字符集可能还互不兼容。举例来说，对于那些使用拉丁字母表的欧洲语言，所用的字符几乎能放进一个256个字符的字符集。不幸的是，还是为不同语言使用了不同的集合，有些不同字符最终得到的是同一个整数值。例如，法文（用Latin1）就不能与冰岛文（用Latin2）共存。能在一个字符集里展示人类所知的所有字符的宏伟尝试已经提供了很大的帮助，但是，即使是16位字符集（例如Unicode）也不足以使每个人都满意。或许能保存每一个字符的32位字符集（如我所知）尚未广泛使用。

简而言之，C++的方式是允许程序员使用任何字符集作为串的字符类型，也可以使用某种扩充的字符集或者可移植的数值编码（C.3.3节）。

20.2.1 字符特征类——*char_traits*

正如13.2节所述，原则上说，串能以任何带有正确的复制操作的类型作为其字符类型。然而，对于那些没有用户定义复制操作的类型，它可以改进效率并简化实现。因此，标准*string*要求作为其字符类型的类型不包含用户定义复制操作，这也有助于串I/O的简化与高效率。

一个字符类型的性质由其*char_traits*（字符特征类）定义。一个*char_traits*就是下面模板的一个专门化：

```
template<class Ch> struct char_traits { };
```

所有的*char_traits*都定义在名字空间*std*里，标准的字符特征类由 *<string>* 给出。通用*char_traits*本身并无任何属性，只有针对特定字符类型的专门*char_traits*具有属性。现在考虑下面的*char_traits<char>*：

```
template<> struct char_traits<char> {      // char_traits操作不应抛出异常
    typedef char char_type;                // 字符类型
    static void assign(char_type&, const char_type&);      // char_type的 =
    // 字符的整数表示：
    typedef int int_type;                  // 字符的整数值类型
    static char_type to_char_type(const int_type&);        // int到char的转换
    static int_type to_int_type(const char_type&);         // char到int的转换
    static bool eq_int_type(const int_type&, const int_type&); // ==
    // char_type比较：
    static bool eq(const char_type&, const char_type&);    // ==
    static bool lt(const char_type&, const char_type&);    // <
    // 对s[n]数组的操作：
    static char_type* move(char_type* s, const char_type* s2, size_t n);
    static char_type* copy(char_type* s, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static int compare(const char_type* s, const char_type* s2, size_t n);
    static size_t length(const char_type*);
    static const char_type* find(const char_type* s, int n, const char_type&);
    // I/O相关操作：
    typedef streamoff off_type;            // 流中的偏移量
```

```

typedef streampos pos_type;    // 流中位置
typedef mbstate_t state_type;  // 多字节流状态

static int_type eof();          // 文件结束
static int_type not_eof(const int_type& i); // i不等于eof()则为i; 否则返回任何非eof()值
static state_type get_state(pos_type p);    // p中字符的多字节转换状态
};

```

标准串模板的实现**basic_string**（20.3节）依赖于这些类型和函数。一个类型要想作为**basic_string**的字符类型，就必须提供支持所有上述功能的专门化的**char_traits**。

一个类型能要作为一个**char_type**，就必须能获得对每个字符的整数值。这个整数值类型为**int_type**，它与**char_type**之间的转换通过**to_char_type()**和**to_int_type()**完成。对于**char**而言，这些转换都是直截了当的。

操作**move(s, s2, n)**和**copy(s, s2, n)**都使用**assign(s[i], s2[i])**从**s2**复制**n**个字符到**s**，两者之间的差异在于，即使**s2**位于**[s, s+n]**的范围内，操作**move()**也能正确工作；而**copy()**的速度可能更快一些。这模仿了标准的C库函数**memcpy()**和**memmov()**（19.4.6节）。调用**assign(s, n, x)**将使用**assign(s[i], x)**，把**n**个**x**复制到**s**里。

compare()函数用**lt()**和**eq()**做字符的比较。它返回**int**值，用**0**表示正好匹配，用负数表示其第一个参数按照字典序先于第二个参数，用正数表示其第一个参数按字典序后于第二个参数。返回值的这种用法是模仿标准C库函数**strcmp()**（20.4.1节）。

与I/O相关的函数在底层I/O的实现中使用（21.6.4节）。

宽字符（即类型为**wchar_t**（4.3节）的对象）也很像**char**，除了它占用两个或者多个字节之外。**wchar_t**类型的性质由**char_traits<wchar_t>**描述：

```

template<> struct char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    typedef wstreamoff off_type;
    typedef wstreampos pos_type;

    // 像char_traits<char>
};

```

wchar_t通常是用于保存16位字符集，如Unicode。

20.3 基础串类——**basic_string**

标准库串功能的基础是模板**basic_string**，它提供了许多成员类型和操作，与标准容器（16.3节）所提供的类似：

```

template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class std::basic_string {
public:
    // ...
};

```

这个模板及其相关功能都定义在名字空间**std**里，由**<string>**给出。

采用了两个**typedef**，为最常用的串类型提供了方便的名字：

```

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;

```

*basic_string*很像*vector* (16.3节), 但是*basic_string*还提供了一些典型的串操作, 例如子串检索, 也没有提供*vector*那样的一组完整操作。*string*不大会直接用数组或者*vector*实现, 为了很好地支持*string*的许多最常见应用, 实现中需要尽量减少复制, 对短的字符串不使用自由存储空间, 允许对长串的简单修改等 (见20.6[12])。 *string*函数的数目反应出串操作的重要性。还有一个事实, 有些机器也为串操作提供了特殊的硬件指令。如果存在具有类似语义的标准库函数, 这些硬件功能将使库的实现者受益匪浅。

与其他标准库类型一样, *basic_string*<*T*> 是一个没有虚函数的具体类型 (2.5.3节、10.3节)。当设计更复杂的文字处理类时, 它可以被用做成员, 但它并无意作为派生类的基类 (25.2.1节、20.6[10])。

20.3.1 类型

与*vector*一样, *basic_string*提供了一组成员类型名, 使人能使用这些与串相关的类型:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // 类型 (很像vector、list等, 16.3.1节);
    typedef Tr traits_type;           // 特别针对basic_string

    typedef typename Tr::char_type value_type;
    typedef A allocator_type;
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef typename A::reference reference;
    typedef typename A::const_reference const_reference;
    typedef typename A::pointer pointer;
    typedef typename A::const_pointer const_pointer;

    typedef implementation_defined iterator;
    typedef implementation_defined const_iterator;

    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    // ...
};
```

*basic_string*的概念除了支持简单的*basic_string*<*char*> (即*string*) 之外, 还支持许多不同种类的字符的串。例如:

```
typedef basic_string<unsigned char> Ustring;

struct Jchar ( /* ... */ );           // 日文字符类型
typedef basic_string<Jchar> Jstring;
```

只要这种字符提供了同样的语义, 相应的串就可以像*char*的串一样使用。例如,

```
Ustring first_word(const Ustring& us)
{
    Ustring::size_type pos = us.find(' '); // 见20.3.11节
    return Ustring(us, 0, pos);           // 见20.3.4节
}

Jstring first_word(const Jstring& js)
{
    // ...
}
```

```

    Jstring::size_type pos = js.find( ' ' ); // 见20.3.11节
    return Jstring( js, 0, pos ); // 见20.3.4节
}

```

当然，同样可以使用以字符串为参数的模板：

```

template<class S> S first_word(const S& s)
{
    typename S::size_type pos = s.find( ' ' ); // 见20.3.11节
    return S( s, 0, pos ); // 见20.3.4节
}

```

basic_string<Ch> 能存放集合**Ch**中的任何字符，特别是**string**里可以有**0**。“字符类型”**Ch**的行为必须像字符，特别是它不能有用户确定的复制构造函数、析构函数和复制赋值。

20.3.2 迭代器

与其他容器一样，一个**string**也提供了常规的和反向的迭代器：

```

template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // 迭代器（类似vector、list等，16.3.2节）：
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;

    // ...
};

```

因为**string**具有获取迭代器所需的成员类型和函数，所以**string**可以与标准算法一起使用（第18章）。例如，

```

void f(string& s)
{
    string::iterator p = find(s.begin(), s.end(), 'a');
    // ...
}

```

对于**string**最常用的操作则是由**string**类直接提供的。当然，有关操作的这些版本将专门为**string**优化，这样做远比对通用操作去做更容易些。

对串而言，标准算法（第18章）并不像人们可能设想的那么有用。通用算法倾向于假设容器的元素具有独立的意义，而对串而言，这一假设多半不成立。一个串的意义融合在其字符的确切序列之中，这样，对一个串排序（对串中字符排序）也就破坏了它的意义，虽然对一般的容器排序常常能使其更为有用。

string迭代器不做范围检查。

20.3.3 元素访问

可以通过下标访问`string`里的各个字符：

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // 元素访问（类似vector，16.3.3节）：
    const_reference operator[] (size_type n) const; // 不检查的访问
    reference operator[] (size_type n);

    const_reference at(size_type n) const;           // 带检查的访问
    reference at(size_type n);

    // ...
};
```

超范围访问将导致`at()`抛出`out_of_range`异常。

与`vector`相比，`string`缺少`front()`和`back()`。要访问一个`string`的第一个和最后一个元素，我们必须分别写`s[0]`和`s[s.length() - 1]`。指针和数组的等价关系对`string`也不成立，如果`s`是个`string`，`&s[0]`和`s`是不一样的。

20.3.4 构造函数

`string`有一组初始化和复制操作，它们与其他容器类所提供的类似东西（16.3.4节）有许多细节上的差异：

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // 构造函数等（有点像vector和list，16.3.4节）：
    explicit basic_string(const A& a = A());
    basic_string(const basic_string& s,
                 size_type pos = 0, size_type n = npos, const A& a = A());
    basic_string(const Ch* p, size_type n, const A& a = A());
    basic_string(const Ch* p, const A& a = A());
    basic_string(size_type n, Ch c, const A& a = A());
    template<class In> basic_string(In first, In last, const A& a = A());

    ~basic_string();

    static const size_type npos; // 表示“所有字符”

    // ...
};
```

要初始化一个`string`，可以通过一个C风格字符串、另一个`string`、C风格字符串的一部分、另一个`string`的一部分或者其他的字符序列。当然，`string`不能用字符或者整数去初始化：

```
void f(char* p, vector<char>&v)
{
    string s0;           // 空串
    string s00 = "";     // 也是空串
    string s1 = 'a';     // 错误：没有从char到string的转换
}
```



```

string s2 = 7;           // 错误：没有从int到string的转换
string s3(7);           // 错误：没有取一个整数参数的构造函数
string s4(7, 'a');       // 'a'的7个副本："aaaaaaa"

string s5 = "Frodo";     // "Frodo"的副本
string s6 = s5;          // s5的副本

string s7(s5, 3, 2);     // s5[3]和s5[4]，即"do"
string s8(p+7, 3);       // p[7], p[8]和p[9]
string s9(p, 7, 3);      // string(string(p), 7, 3)，可能费时间
string s10(v.begin(), v.end()); // 从v复制所有字符
}

```

字符的编号从0开始，所以string是编号为0到length() - 1的字符的序列。

一个串的length()是它的size()的同义词，两个函数都返回串中字符的个数。请注意，这些函数不能用于统计C风格的以0结束的字符串。在basic_string的实现中存储着长度，并不依赖于结束符。

了串采用字符位置加上字符个数表示。npos的默认值被初始化为最大的可能值，用于表示“所有成员”。

不存在创建n个不确定字符的串的构造函数，与此最接近的是做出一个包含n个同一给定字符的构造函数。没有只以一个字符做参数的构造函数，也没有只提供字符个数的构造函数，这就使编译器能够检查出上面s2和s3那样的定义错误。

复制构造函数是带有四个参数的构造函数，其中的三个有默认值。为了提高效率，也可以将这个构造函数实现为两个独立的构造函数，用户如果不去看实际生成的代码，就不会知道其中的情况。

上面声明里的那个模板形式的构造函数是最一般的，它使我们可以用任意序列的值去初始化一个串。特别是允许用不同字符集的元素去初始化一个串，只要存在着字符间的转换。例如：

```

void f(string s)
{
    wstring ws(s.begin(), s.end()); // 从s复制所有的字符
    // ...
}

```

ws里的每个wchar_t都是来自s的对应char初始化的。

20.3.5 错误

字符串经常用于读入、写出、打印、存储、比较、复制等。这不会引起什么问题，最糟也就是出现某些性能问题。然而，一旦我们开始去操纵个别的子串和字符，从已有字符串去组成新串，那么迟早我们会犯错误，以至于写到了串的端点之外。

对于显式地访问个别的字符，at()将进行检查，如果我们试图在串端点之外访问，它就会抛出out_of_range异常；[]则不检查。

许多串操作以字符位置加上字符个数作为参数。如果所给的位置大于串的大小，它们就抛出out_of_range异常，“过大”的字符计数值则简单地当做取出这个串“剩余”的字符。例如，

```

void f()
{
    string s = "Snobol4";
    string s2(s, 100, 2); // 字符位置超出串尾, 抛出out_of_range
    string s3(s, 2, 100); // 字符计数值过大, 等价于s3(s, 2, s.size() - 2)
    string s4(s, 2, string::npos); // 字符串从s[2]开始
}

```

可见, “过大”的位置值必须避免, 而“过大”的字符计数值则可以使用。事实上, *npos*就是*size_type*的最大可能值。

我们试试给一个负的位置或者字符计数值:

```

void g(string& s)
{
    string s5(s, -2, 3); // 位置太大! 抛出out_of_range
    string s6(s, 3, -2); // 字符计数值大! 可以
}

```

因为表示位置和计数值的*size_type*是*unsigned*类型, 所以一个负数就被糊里糊涂地当做了一个很大的正数了(16.3.4节)。

注意那些用于找出*string*中的子串的函数(20.3.11节)。在没有找到子串时, 它们返回*npos*, 这样它们就不抛出异常了。然而, 如果后面以*npos*作为字符位置, 那就会抛出异常。

刻画子串的另一种方式是使用一对迭代器, 第一个迭代器表示字符位置, 两个迭代器之差作为字符计数值。通常, 迭代器都不做范围检查。

在使用C风格的字符串时, 检查范围就会变得更加困难。如果给了一个C风格字符串(一个指向*char*的指针), *basic_string*的函数都假定这个指针不是0。如果给的是C风格字符串中的某个位置, 它们都假定这个C风格字符串足够长, 使这里是一个合法的位置。要当心! 在这里, 当心的意思就是极度小心, 除了在使用字符串文字量时。

对于所有的串, *length() < npos*都成立。在很少几种情况下, 例如将一个字符串插入另一个(20.3.9节), 也有可能(虽然不大可能)构造出一个因为过长而无法表示的串。在这种情况下将抛出*length_error*。例如:

```

string s(string::npos, 'a'); // 抛出length_error

```

20.3.6 赋值

很自然, 为串提供了赋值操作:

```

template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch>>
class basic_string {
public:
    // ...
    // 赋值(有点像vector和list, 16.3.4节);
    basic_string& operator=(const basic_string& s);
    basic_string& operator=(const Ch* p);
    basic_string& operator=(Ch c);

    basic_string& assign(const basic_string&);
    basic_string& assign(const basic_string& s, size_type pos, size_type n);
    basic_string& assign(const Ch* p, size_type n);
    basic_string& assign(const Ch* p);
    basic_string& assign(size_type n, Ch c);
}

```

```

        template<class In> basic_string& assign(In first, In last);
        // ...
};

```

与其他容器一样，*string*也采用值语义，也就是说，在将一个串赋值给另一个串时，被赋值的串将被复制，赋值之后存在着两个相互独立的同样的串。看下面例子：

```

void g()
{
    string s1 = "Knold";
    string s2 = "Tot";

    s1 = s2;           // 两个 "Tot" 的副本
    s2[1] = 'u';       // s2是 "Tut", s1仍为 "Tot"
}

```

虽然不能用单个字符对串做初始化，但允许用单个的字符给串赋值：

```

void f()
{
    string s = 'a'; // 错误：用char初始化
    s = 'a';        // 可以：赋值
    s = "a";
    s = s;
}

```

能够用单个*char*给*string*赋值实际上并没多大用处，而且通常都认为容易引起错误。当然，因为用 += 附加*char*有时被当做必不可少的操作（20.3.9节），如果允许写 *s* += 'c' 但不允许 *s* = 'c' 就会令人感到奇怪了。

名字*assign()*用于赋值，它对应于多参数的构造函数（16.3.4节、20.3.4节）。

正如在11.12节中所提到的，完全有可能去优化*string*，使得在真正需要*string*的两个副本之前并不做实际复制。标准库*string*的设计也鼓励这类尽可能减少复制的实现方式。采用这种实现技术，以只读方式使用串和将串作为参数传递给函数都将变得非常廉价，其收获将远远超出简单的想象。当然，如果程序员想去写代码，依赖于*string*复制已经做过优化，但在此之前却没有去检查他们所用的实现，那就是过于幼稚了（20.6[13]）。

20.3.7 到C风格字符串的转换

如20.3.4节所述，可以用C风格的字符串去初始化*string*，也可以用C风格字符串给*string*赋值。反过来说，也可以将一个*string*中所有字符的一个副本放进数组：

```

template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // 转换到C风格字符串:

    const Ch* c_str() const;
    const Ch* data() const;
    size_type copy(Ch* p, size_type n, size_type pos = 0) const;
    // ...
};

```

函数*data()*将*string*里的字符写进数组，并返回指向这个数组的指针。这个数组由*string*所拥

有，用户不应试图去删除它。在随后对串的非`const`操作之后，用户也不能依赖这个数组的值。`c_str()`函数很像`data()`，只是它在最后加上一个0作为C风格字符串的结束符。例如，

```
void f()
{
    string s = "equinox";           // s.length() == 7
    const char* p1 = s.data();       // p1指向7个字符
    printf("p1 = %s\n", p1);         // 坏的：缺少结束符
    p1[2] = 'a';                     // 错误：p1指向const数组
    s[2] = 'a';
    char c = p1[1];                  // 坏的：在修改s后访问s.data()

    const char* p2 = s.c_str();       // p2指向8个字符
    printf("p2 = %s\n", p2);         // 可以：c_str()加了结束符
}
```

换句话说，`data()`产生的是字符数组，而`c_str()`产生的是C风格字符串。这几个函数基本上是为了允许使用那些以C风格字符串为参数的函数，因此`c_str()`通常比`data()`更有用一些。例如，

```
void f(string s)
{
    int i = atoi(s.c_str());         // 取得串中数字形成的int值（20.4.1节）
    // ...
}
```

一般说，最好还是让字符保存在`string`里，一直到你需要用它们的时候。当然，如果你不能一下子就用完这些字符，你也可能将它们复制到一个数组里，不应该让它们留在由`c_str()`或`data()`分配的缓冲区中。`copy()`函数就是为此而提供的，例如，

```
char* c_string(const string& s)
{
    char* p = new char[s.length()+1]; // 注意：+1
    s.copy(p, string::npos);
    p[s.length()] = 0;                 // 注意：加结束符
    return p;
}
```

调用`s.copy(p, n, m)`从`s[m]`开始复制至多`n`个字符到`p`。如果在`s`里能够复制的字符少于`n`，`copy()`就复制那里的所有字符。

请注意，`string`也可以包含0字符。操作C风格字符串的那些函数都将把0字符解释为结束符。在将0字符放进字符串里时请一定当心，只在你不使用C风格的字符串函数，或者用0字符就是作为结束符时才可以这样做。

到C风格字符串的转换原本也可以通过定义`operator const char*()`的方式提供，而不是通过`c_str()`。采用这样的做法能带来显式转换上的方便，但付出的代价将是在某些情况下引起令人惊诧的不期望的隐式转换。

如果你发现在自己的程序里`c_str()`频繁出现，那可能是因为你过于依赖于C风格的界面。通常也有依赖于`string`（而不是C风格字符串）的界面可以用，利用它就可以清除这些转换。换另一种方式，如果某些函数迫使你经常去写`c_str()`，你也可以通过为它们提供附加定义的方式避免大部分对`c_str()`的显式调用：

```
extern "C" int atoi(const char*);
```

```
int atoi(const string& s)
{
    return atoi(s.c_str());
}
```

20.3.8 比较

串可以与类型相同的串进行比较,也可以与具有同样字符类型的数组比较:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...

    int compare(const basic_string& s) const; // 组合>和==
    int compare(const Ch* p) const;

    int compare(size_type pos, size_type n, const basic_string& s) const;
    int compare(size_type pos, size_type n,
                const basic_string& s, size_type pos2, size_type n2) const;
    int compare(size_type pos, size_type n, const Ch* p, size_type n2 = npos) const;

    // ...
};
```

如果在对一个**string**使用**compare()**时提供了一个位置和一个大小,那么比较中就只用指定的子串。例如,**s.compare(pos, n, s2)**等价于**string(s, pos, n).compare(s2)**。使用做比较的准则是**char_traits<Ch>**的**compare()**(20.2.1节),这样,如果串值相同,**s.compare(s2)**返回0;如果**s**按照字典序先于**s2**,它返回一个负值;否则就返回一个正值。

用户可以按13.4节的方式提供比较准则。如果需要更强的灵活性,我们可以去使用**lexicographical_compare()**(18.9节),定义一个类似13.4节里那样的函数,或者显式地写一个循环。例如,**toupper()**函数(20.4.2节)使我们可以写出如下不考虑大小写的比较函数:

```
int cmp_nocase(const string& s, const string& s2)
{
    string::const_iterator p = s.begin();
    string::const_iterator p2 = s2.begin();

    while (p != s.end() && p2 != s2.end()) {
        if (toupper(*p) != toupper(*p2)) return (toupper(*p) < toupper(*p2)) ? -1 : 1;
        ++p;
        ++p2;
    }

    return (s2.size() == s.size()) ? 0 : (s.size() < s2.size()) ? -1 : 1; // 大小并无符号
}

void f(const string& s, const string& s2)
{
    if (s == s2) { // 比较s与s2,考虑大小写
        // ...
    }

    if (cmp_nocase(s, s2) == 0) { // 比较s与s2,不考虑大小写
        // ...
    }

    // ...
}
```

对 `basic_string` 也提供了普通的比较运算符 `==`、`!=`、`>`、`<`、`>=`、`<=`：

```
template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch, Tr, A>&, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
bool operator==(const Ch*, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
bool operator==(const basic_string<Ch, Tr, A>&, const Ch*);

// 对 !=、>、<、>=、<= 的声明类似
```

比较运算符为非成员函数，因此对两个运算对象都同样可以实施转换（11.2.3节）。这里还提供了取C风格字符串为参数的版本，以优化与字符串文字量的比较。例如，

```
void f(const string& name)
{
    if (name == "Obelix" || "Asterix" == name) {    // 使用优化的 ==
        // ...
    }
}
```

20.3.9 插入

一旦建立了一个串之后，就可以以许多不同的方式对它进行操作了。在所有修改串值的操作中，最常用的就是对它做附加，也就是说，将一些字符加到串的最后。在其他位置的插入则比较罕见：

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch>>
class basic_string {
public:
    // ...
    // 在 (*this)[length()-1] 之后添加字符：
    basic_string& operator+=(const basic_string& s);
    basic_string& operator+=(const Ch* p);
    basic_string& operator+=(Ch c);
    void push_back(Ch c);

    basic_string& append(const basic_string& s);
    basic_string& append(const basic_string& s, size_type pos, size_type n);
    basic_string& append(const Ch* p, size_type n);
    basic_string& append(const Ch* p);
    basic_string& append(size_type n, Ch c);
    template<class In> basic_string& append(In first, In last);

    // 在 (*this)[pos] 之前添加字符：
    basic_string& insert(size_type pos, const basic_string& s);
    basic_string& insert(size_type pos, const basic_string& s, size_type pos2, size_type n);
    basic_string& insert(size_type pos, const Ch* p, size_type n);
    basic_string& insert(size_type pos, const Ch* p);
    basic_string& insert(size_type pos, size_type n, Ch c);

    // 在 p 之前添加字符：
    iterator insert(iterator p, Ch c);
    void insert(iterator p, size_type n, Ch c);
    template<class In> void insert(iterator p, In first, In last);
```

```
// ...
};
```

简而言之，针对串初始化和给串赋值的每个操作，对应的都有附加操作和在某个字符位置之前插入的操作。

运算符 += 是做附加操作的最常用的方便形式。例如，

```
string complete_name(const string& first_name, const string& family_name)
{
    string s = first_name;
    s += ' ';
    s += family_name;
    return s;
}
```

在最后附加比在其他地方插入的效率要高得多。例如，

```
string complete_name2(const string& first_name, const string& family_name) // 糟糕的算法
{
    string s = family_name;
    s.insert(s.begin(), ' ');
    return s.insert(0, first_name);
}
```

插入操作常常会迫使string实现去做额外的存储管理操作，并需要来回搬动字符。

因为string也有push_back()操作(16.3.5节)，所以也可以对string使用back_inserter，和其他容器一样。

20.3.10 拼接

附加是拼接的一种特殊情况。拼接就是从两个串出发构造出一个新串，将其中的一个串放在另一个之后，这一操作通过+运算符提供：

```
template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A>
operator+(const basic_string<Ch, Tr, A>&, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+(const Ch*, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+(Ch, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+(const basic_string<Ch, Tr, A>&, const Ch*);

template<class Ch, class Tr, class A>
basic_string<Ch, Tr, A> operator+(const basic_string<Ch, Tr, A>&, Ch);
```

通常，运算符+同样被定义为非成员函数。对那些带有几个模板参数的模板，写时在记法形式上就有些不方便，需要反复地提及这些模板参数。

但在另一方面，拼接的使用则既明晰又方便。例如，

```
string complete_name3(const string& first_name, const string& family_name)
{
    return first_name + ' ' + family_name;
}
```

与`complete_name()`相比,这种写法上的方便可能是通过某些运行时的额外代价交换来的。`complete_name3()`需要另外一块临时存储(11.3.2节)。按照我的经验,这种情况很少产生重大影响,但在写程序中那些性能特别要紧的内层循环时,就应该记住这件事情。在那种情况下,我们甚至应该通过`complete_name()`在线化避免一次函数调用,通过底层操作在当地构造出结果串(20.6[4])。

20.3.11 查找

各种查找子串的函数多得令人有些手足无措:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
public:
    // ...
    // 查找子串 (类似search(), 18.5.5节):
    size_type find(const basic_string& s, size_type i = 0) const;
    size_type find(const Ch* p, size_type i, size_type n) const;
    size_type find(const Ch* p, size_type i = 0) const;
    size_type find(Ch c, size_type i = 0) const;

    // 从末端起反向查找子串 (类似find_end(), 18.5.5节):
    size_type rfind(const basic_string& s, size_type i = npos) const;
    size_type rfind(const Ch* p, size_type i, size_type n) const;
    size_type rfind(const Ch* p, size_type i = npos) const;
    size_type rfind(Ch c, size_type i = npos) const;

    // 查找字符 (类似find_first_of(), 18.5.2节):
    size_type find_first_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_of(const Ch* p, size_type i = 0) const;
    size_type find_first_of(Ch c, size_type i = 0) const;

    // 根据参数从末端起反向查找字符:
    size_type find_last_of(const basic_string& s, size_type i = npos) const;
    size_type find_last_of(const Ch* p, size_type i, size_type n) const;
    size_type find_last_of(const Ch* p, size_type i = npos) const;
    size_type find_last_of(Ch c, size_type i = npos) const;

    // 查找不在参数里的字符:
    size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
    size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_first_not_of(const Ch* p, size_type i = 0) const;
    size_type find_first_not_of(Ch c, size_type i = 0) const;

    // 从末端起反向查找不在参数里的字符:
    size_type find_last_not_of(const basic_string& s, size_type i = npos) const;
    size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
    size_type find_last_not_of(const Ch* p, size_type i = npos) const;
    size_type find_last_not_of(Ch c, size_type i = npos) const;
    // ...
};
```

这些都是`const`成员函数,也就是说,它们只是为了针对某种用途去找出子串的位置,并不修改它们所作用的那个串的值。

basic_string::find函数的意义可以从更一般的算法等价来理解。考虑下面例子：

```
void f()
{
    string s = "accdcde";
    string::size_type i1 = s.find("cd");           // i1 = 2  s[2] == 'c' && s[3] == 'd'
    string::size_type i2 = s.rfind("cd");           // i2 = 4  s[4] == 'c' && s[5] == 'd'
    string::size_type i3 = s.find_first_of("cd");    // i3 = 1  s[1] == 'c'
    string::size_type i4 = s.find_last_of("cd");     // i4 = 5  s[5] == 'd'
    string::size_type i5 = s.find_first_not_of("cd"); // i5 = 0  s[0] != 'c' && s[0] != 'd'
    string::size_type i6 = s.find_last_not_of("cd"); // i6 = 6  s[6] != 'c' && s[6] != 'd'
}
```

如果**find()**没找到任何匹配，它就返回**npos**，表示一个非法的字符位置。如果以**npos**作为字符位置，那么就会抛出**out_of_range**异常（20.3.5节）。

注意，**find()**的结果是一个**unsigned**值。

20.3.12 替换

一旦在某个串里标识出了一个位置，我们就可以通过下标操作修改个别位置的字符值，或者通过**replace()**用一些新字符替换整个子串：

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch>>
class basic_string {
public:
    // ...
    // 用其他字符替换 [(*this)[i], (*this)[i + n]];

    basic_string& replace(size_type i, size_type n, const basic_string& s);
    basic_string& replace(size_type i, size_type n,
                          const basic_string& s, size_type i2, size_type n2);
    basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2);
    basic_string& replace(size_type i, size_type n, const Ch* p);
    basic_string& replace(size_type i, size_type n, size_type n2, Ch c);

    basic_string& replace(iterator i, iterator i2, const basic_string& s);
    basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n);
    basic_string& replace(iterator i, iterator i2, const Ch* p);
    basic_string& replace(iterator i, iterator i2, size_type n, Ch c);
    template<class In> basic_string& replace(iterator i, iterator i2, In j, In j2);

    // 从串中删除（“用空串替换”）：

    basic_string& erase(size_type i = 0, size_type n = npos);
    iterator erase(iterator i);
    iterator erase(iterator first, iterator last);

    // ...
};
```

注意，那些新字符的数目不必与串中原来的字符数目相同。字符串的大小将依据新子串的情况改变。特别地，**erase()**删除有关子串，并调整字符串的大小。例如，

```
void f()
{
    string s = "but I have heard it works even if you don't believe in it";
    s.erase(0, 4);           // 删除开始的 "but"
    s.replace(s.find("even"), 4, "only");
    s.replace(s.find("don't"), 5, ""); // 通过用 "" 替换来删除
}
```

简单地调用`erase()`而不给参数,就会将这个串做成空串。这对应于一般容器中称为`clear()`的操作(16.3.6节)。

`replace()`函数的各种形式与赋值相对应,因为`replace()`是一种对子串的赋值。

20.3.13 子串

`substr()`函数使你能用一个位置加上一个长度去描述子串:

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch>>
class basic_string {
public:
    // ...
    // 子串定位:
    void clear(); // 删除所有字符
    basic_string substr(size_type i = 0, size_type n = npos) const;
    // ...
};
```

`substr()`函数也就是一种读出子串的方式。另一方面,用`replace()`操作可以写入一个子串。这两个操作都依赖于一个低级的位置再加上一个字符个数。当然,还有`find()`函数可以用值去找到子串。利用这些函数,我们就可以定义出一种既能读也能写的子串:

```
template<class Ch> class Basic_substring {
public:
    typedef typename basic_string<Ch>::size_type size_type;

    Basic_substring(basic_string<Ch>& s, size_type i, size_type n); // s[i]..s[i+n-1]
    Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2); // s2在s里
    Basic_substring(basic_string<Ch>& s, const Ch* p); // *p在s里
    Basic_substring& operator=(const basic_string<Ch>&); // 通过 *ps写
    Basic_substring& operator=(const Basic_substring<Ch>&);
    Basic_substring& operator=(const Ch*);
    Basic_substring& operator=(Ch);

    operator basic_string<Ch>() const; // 通过 *ps读
    operator const Ch*() const; // 用c_str()

private:
    basic_string<Ch>* ps;
    size_type pos;
    size_type n;
};
```

其实现基本上是很平凡的。例如,

```
template<class Ch>
Basic_substring<Ch>::Basic_substring(basic_string<Ch>& s, const basic_string<Ch>& s2)
: ps(&s), n(s2.length())
{
    pos = s.find(s2);
}

template<class Ch>
Basic_substring<Ch>& Basic_substring<Ch>::operator=(const basic_string<Ch>& s)
{
    ps->replace(pos, n, s); // 通过 *ps写
    return *this;
}
```

```
template<class Ch> Basic_substring<Ch>::operator basic_string<Ch>() const
{
    return basic_string<Ch>(*ps, pos, n);    // 从 *ps复制
}
```

如果在 s 里没找到 $s2$, pos 就是 $npos$ 。企图去读或写它都将抛出`out_of_range`异常(20.3.5节)。

这个`Basic_substring`可以像下面这样使用:

```
typedef Basic_substring<char> Substring;
void f()
{
    string s = "Mary had a little lamb";
    Substring(s, "lamb") = "fun";
    Substring(s, "a little") = "no";
    string s2 = "Joe" + Substring(s, s.find(' '), string::npos);
}
```

当然, 如果`Basic_substring`能做某些模式匹配, 事情就更有意思了(20.6[7])。

20.3.14 大小和容量

有关存储问题的处理很像`vector`的情况(16.3.8节):

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch>>
class basic_string {
public:
    // ...
    // 大小、容量等(类似16.3.8节):
    size_type size() const;                // 字符个数(20.3.4节)
    size_type max_size() const;            // 最大可能的串
    size_type length() const { return size(); }
    bool empty() const { return size() == 0; }

    void resize(size_type n, Ch c);
    void resize(size_type n) { resize(n, Ch()); }

    size_type capacity() const;            // 类似vector: 16.3.8节
    void reserve(size_type res_arg = 0);    // 类似vector: 16.3.8节

    allocator_type get_allocator() const;
};
```

如果 $res_arg > max_size()$, 调用`reserve(res_arg)`就会抛出`length_error`。

20.3.15 I/O操作

`string`的一项主要用途就是作为输入目标和输出源。`basic_string`的输入和输出操作在`<string>`里提供(不是在`<iostream>`里):

```
template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& operator>>(basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>&, const basic_string<Ch, Tr, A>&);

template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline(basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&, Ch eol);
```

```
template<class Ch, class Tr, class A>
basic_istream<Ch, Tr>& getline(basic_istream<Ch, Tr>&, basic_string<Ch, Tr, A>&);
```

<< 运算符将一个串写入一个 *ostream* (21.2.1节)。>> 运算符将一个以空白结束的完整的字读入 (3.6节、21.3.1节) 到对应字符串里, 并在必要时为保存这个字而扩充字符串。开头的空白字符将被跳过, 结束处的空白字符也不放进字符串里。

getline() 函数将由 *eol* 结束的一个完整行读入它的字符串, 并可能扩展字符串以存放这个行 (3.6节)。如果没有提供 *eol* 参数, 那么就换行符 '\n' 作为结束符。这个行结束符被从流中去除, 但也不放入串里。因为 *string* 能够为保存输入而自动扩展, 因此就不必将这种结束符留在流里, 也不需要提供字符计数值 (像为字符数组所用的 *get()* 或者 *getline()* 那样, 21.3.4节)。

20.3.16 交换

与 *vector* (16.3.9节) 类似, 专门针对 *string* 的函数可能做得比一般算法的效率 high 许多, 因此提供了一个特殊版本:

```
template<class Ch, class Tr, class A>
void swap(basic_string<Ch, Tr, A>&, basic_string<Ch, Tr, A>&);
```

20.4 C标准库

C++ 标准库从 C 标准库继承了一批 C 风格字符串函数。本节列出其中的一些最有用的 C 字符串函数, 这个描述并不完全, 相关的进一步信息请查阅你的参考手册。请留意, 实现者常常在标准头文件里添加一些他们自己的非标准函数, 结果很容易把人搞糊涂, 弄不清到底哪些函数在每个实现里都能用。

给出各种标准 C 库功能的头文件都列在 16.1.2 节里。存储管理函数可以在 19.4.6 节找到, C 语言 I/O 函数在 21.8 节, C 语言数学库在 22.3 节。与启动和终止有关的函数在 3.2 节和 9.4.1.1 节描述, 读入未予刻画的函数参数的机制在 7.6 节介绍。为处理宽字符的串的 C 风格函数可以在 *<wchar>* 和 *<wchar.h>* 中找到。

20.4.1 C风格字符串

操作 C 风格字符串的函数可以在 *<string.h>* 和 *<cstring>* 里找到:

```
char* strcpy(char* p, const char* q);           // 从q复制到p (包括结束符)
char* strcat(char* p, const char* q);           // 将q附加到p后 (包括结束符)
char* strncpy(char* p, const char* q, int n);    // 从q复制n个字符到p (包括结束符)
char* strncat(char* p, const char* q, int n);    // 将q的n个字符附加到p后 (包括结束符)
size_t strlen(const char* p);                   // p的长度 (不计结束符)

int strcmp(const char* p, const char* q);         // 比较: p和q
int strncmp(const char* p, const char* q, int n); // 比较前n个字符

char* strchr(char* p, int c);                    // 在p中找第一个c
const char* strchr(const char* p, int c);         // 在p中找第一个c
char* strrchr(char* p, int c);                   // 在p中找最后一个c
const char* strrchr(const char* p, int c);        // 在p中找最后一个c
char* strstr(char* p, const char* q);             // 在p中找第一个q
const char* strstr(const char* p, const char* q); // 在p中找第一个q
```

```
char* strpbrk(char* p, const char* q); // 在p中找q的第一个字符
const char* strpbrk(const char* p, const char* q);

size_t strspn(const char* p, const char* q); // p中出现不属于q的字符之前的字符个数
size_t strcspn(const char* p, const char* q); // p中出现属于q的字符之前的字符个数
```

这里都假定指针非零,假定它所指的`char`数组里的字符由`0`结束。在需要复制的字符不足`n`个时,`strn`函数将加上一些`0`。字符串比较在串相等时返回`0`,如果第一个参数按照字典序先于第二个时返回某个负数,否则返回某个正数。

自然,C原本并没有提供任何重载函数,但是在C++里为了`const`安全性,在这里就需要有重载。例如,

```
void f(const char* pcc, char* pc) // C++
{
    *strchr(pcc, 'a') = 'b'; // 错误: 不能给const char赋值
    *strchr(pc, 'a') = 'b'; // 可以,但草率: p里可能没有'a'
}
```

C++的`strchr()`不允许你向`const`里写。然而,C程序也完全可能“利用”在C的`strchr()`里较弱的类型检查:

```
char* strchr(const char* p, int c); /* C标准库函数,但不是C++的 */
void g(const char* pcc, char* pc) /* C程序,在C++里无法编译 */
{
    *strchr(pcc, 'a') = 'b'; /* 将const转换为非const: 在C里可以,在C++里是错误 */
    *strchr(pc, 'a') = 'b'; /* 在C和C++里均可 */
}
```

只要可能,应该始终避免C风格的字符串,尽量使用`string`。C风格字符串及其函数可以用于产生效率极高的代码,但即使是有经验的C和C++程序员,在使用它们时,也常常做出一些无法捕捉的“无聊错误”。当然,C++程序员都无法避免在老代码里看到这些函数。这里是一个并无特别意义的例子,只是为了展示最常用的函数:

```
void f(char* p, char* q)
{
    if (p==q) return; // 指针相等
    if (strcmp(p,q)==0) { // 字符串值相等
        int i = strlen(p); // 字符个数(不包括结束符)
        // ...
    }
    char buf[200];
    strcpy(buf,p); // 将p复制到buf(包括结束符)
    // 草率: 总有一天会溢出
    strncpy(buf,p,200); // 将200个字符从p复制到buf
    // 草率: 总有一天因为复制结束符而出问题
    // ...
}
```

C风格字符串的输入输出通常是通过`printf`一族函数完成的(21.8节)。

在`<stdlib.h>`和`<cstdlib>`里,标准库还提供了几个很有用的将表示数值的字符串转换到数值的函数。例如,

```
double atof(const char* p); // 将p[]转换到double (“alpha to double”)
double strtod(const* p, char**end); // 将p[]转换到double (“string to double”)
```

```
int atoi(const char* p);           // 将p[] 转换到int, 假定基数为10
long atol(const char* p);         // 将p[] 转换到long, 假定基数为10
long strtol(const char* p, char** end, int b); // 将p[] 转换到long, 假定基数为b
```

这些函数将忽略开头的空白。如果对应字符串并不代表一个数, 那么就返回0。例如, `atoi("seven")` 的值就是0。

如果在调用`strtol(p, end, b)`时的`end`不是0, `*end`将被设置到输入串中第一个未读的字符的位置, 使之可用。如果`b == 0`, 数将按照C++整数文字量的形式(4.4.1节)解释; 例如, `0x`前缀表示是十六进制数, `0`前缀表示是八进制数等。

如果`atof()`、`atoi()`或`atol()`转换出的值在各自指定的返回类型中无法表示, 这时发生的情况没有定义。如果`strtol()`的输入串表示的是一个数, 但它无法在`long int`中表示, 或者`strtod()`的输入串表示的数无法在`double`里表示, 那么`errno`(16.1.2节、22.3节)将设置为`ERANGE`, 并返回一个适当的大值或者小值。

除了有关错误处理的方式不同外, `atof(s)`等价于`strtod(s, 0)`, `atoi(s)`等价于`int(strtol(s, 0, 10))`, 而`atol(s)`等价于`strtol(s, 0, 10)`。

20.4.2 字符分类

在`<ctype.h>`和`<cctype>`里, 标准库提供了一组很有用的函数, 用于处理ASCII或者其他类似的字符集:

```
int isalpha(int); // 字母: 在C现场里是'a'..'z' 'A'..'Z' (20.2.1节、21.7节)
int isupper(int); // 大写字母: 在C现场里是'A'..'Z' (20.2.1节、21.7节)
int islower(int); // 小写字母: 在C现场里是'a'..'z' (20.2.1节、21.7节)
int isdigit(int); // 十进制数字: '0'..'9'
int isxdigit(int); // 十六进制数字: '0'..'9' 或 'a'..'f' 或 'A'..'F'
int isspace(int); // ' ' '\t' '\n' 回车符 换行符 换页符
int iscntrl(int); // 控制字符 (ASCII 0..31及127)
int ispunct(int); // 标点符号 (上面字符之外)
int isalnum(int); // isalpha() | isdigit()
int isprint(int); // 可打印字符: ASCII ' ' .. '~'
int isgraph(int); // isalpha() | isdigit() | ispunct()

int toupper(int c); // c对应的大写字母
int tolower(int c); // c对应的小写字母
```

这些通常都通过简单的查找实现, 用字符作为到一个字符属性表的下标。这也意味着如下形式的结构:

```
if (('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')) { // 字母
    // ...
}
```

是低效的, 写起来很讨厌, 而且可能是错误的 (在使用EBCDIC字符集的机器上, 这样做将会接受非字母字符)。

上述函数都以`int`作为参数, 传递给它们的整数必须能表示为`unsigned char`或者`EOF` (最常见是用`-1`表示)。在`char`用带符号表示的系统中, 这也会成为一个问题 (20.6[11])。

针对宽字符的类似函数可以在`<cwctype>`和`<wctype.h>`里找到。

20.5 忠告

[1] 尽量使用`string`操作, 少用C风格字符串函数; 20.4.1节。

- [2] 用`string`作为变量或者成员，不作为基类；20.3节、25.2.1节。
- [3] 你可以将`string`作为参数值或者返回值，让系统去关心存储管理问题；20.3.6节。
- [4] 当你希望做范围检查时，请用`at()`而不是迭代器或者`[]`；20.3.2节、20.3.5节。
- [5] 当你希望优化速度时，请用迭代器或`[]`而不是`at()`；20.3.2节、20.3.5节。
- [6] 直接或者间接地使用`substr()`去读子串，用`replace()`去写子串；20.3.12节、20.3.13节。
- [7] 用`find()`操作在`string`里确定值的位置（而不是写一个显式的循环）；20.3.11节。
- [8] 在你需要高效率地添加字符时，请在`string`的后面附加；20.3.9节。
- [9] 在没有极端时间要求情况下用`string`作为字符输入的目标；20.3.15节。
- [10] 用`string::npos`表示“`string`的剩余部分”；20.3.5节。
- [11] 如果必要，就采用低级操作去实现极度频繁使用的`string`（而不是到处用低级数据结构）；20.3.10节。
- [12] 如果你使用`string`，请在某些地方捕捉`length_error`和`out_of_range`异常；20.3.5节。
- [13] 小心，不要将带值0的`char*`传递给字符串函数；20.3.7节。
- [14] 只是到必须做的时候，（再）用`c_str()`产生`string`的C风格表示；20.3.7节。
- [15] 当你需要知道字符的类别时，用`isalpha()`、`isdigit()`等函数，不要自己去写对字符值的检测；20.4.1节。

20.6 练习

对本章中的一些练习，通过查看标准库实现的源文件可以找到有关的解决方法。为你自己着想：在查看你所用的库的实现者如何解决这些问题之前，先试着自己去找出一些解决方法。

1. (*2) 写一个函数，它取得两个`string`并返回一个`string`，返回值是参数的拼接，在中间加了一个圆点。例如，给的是`file`和`write`，函数返回`file.write`。对C风格的字符串做同样练习，只用C的功能，如`malloc()`和`strlen()`等。比较两个函数，做这类比较的合理准则是什么？
2. (*2) 请列出`vector`和`basic_string`各方面的差异。那些差异是最重要的？
3. (*2) 串的功能并不很规范。例如，你可以用`char`给串赋值，但却不能用`char`做`string`的初始化。列出一个有关这种不规范性的表，其中的哪些可以清除掉而又不会使串的使用复杂化？这样做会引进新的不规范性吗？
4. (*1.5) 类`basic_string`有大量的成员函数，其中哪一些可以做成非成员函数而又不会在效率和书写方便性上造成损害？
5. (*1.5) 写出一个能在`basic_string`上工作的`back_inserter()`版本（19.2.4节）。
6. (*2) 完成20.3.13节中的`Basic_substring`，将它与一个`String`类型集成起来，该类型通过重载`()`表示“子串”，其他方面的行为与`string`一样。
7. (*2.5) 写一个`find()`函数，它在`string`里查找与一种简单的正则表达式的第一个匹配。用`?`表示任意字符，用`*`表示不能与正则表达式后面部分匹配的任意个字符，用`[abc]`表示在方括号中的任意字符（这里是`a`、`b`和`c`），其他字符只与自己匹配。例如，`find(s, "name: ")`返回指向`name:`在`s`中第一次出现的指针；`find(s, [nN]ame:)`返回指向`s`中`name:`或`Name:`的第一次出现的指针；`find(s, "[nN]ame(*)")`返回指向`s`中，后跟括起的任意字符序列（可以为空）的`Name`或者`name`第一次出现的指针。
8. (*2.5) 你认为20.6[7]题的简单正则表达式函数缺少什么操作？描述并添加它们，将你的正

则表达式的描述能力与某个广泛流行和使用的正则表达式的描述能力做一个比较。将你的正则表达式的性能与某个广泛流行使用的正则表达式的性能做一个比较。

9. (*2.5) 利用一个正则表达式库，为一个具有相关的**Substring**类的**String**类实现模式匹配操作。
10. (*2.5) 考虑写一个“理想的”通用文字处理类，称其为**Text**。它应该具有什么功能？你的这一组“理想”功能将给实现强加什么样的约束和额外开销？
11. (*1.5) 定义一组重载的**isalpha()**、**isdigit()**等，使这些函数对**char**、**unsigned char**和**signed char**都能正确工作。
12. (*2.5) 写一个特别为不多于8个字符的字符串优化的**String**类。将它的执行与11.12节的**String**，以及你所用实现的标准库**string**版本做一些比较。有可能设计出一个串类，使之能为很短的字符串所做的优化与完全一般性的字符串组合到一起吗？
13. (*2) 实测**string**复制的性能。你所用实现里**string**的实现复制做了适当优化吗？
14. (*2.5) 比较20.3.9节和20.3.10节的三个**complete_name**版本的性能。试着去写一个运行得尽可能快的版本，记录下在实现和测试过程中发现的所有错误。
15. (*2.5) 假设从**cin**读一些中等长度的串（大约5到25个字符长）是你所用系统里的瓶颈。写一个输入函数，它读这种串的速度能达到你所能设想的最快的程度。你可以选择某种函数界面使之能优化速度，不管使用的方便性。将你做的结果与你所用的实现里的**string**进行比较。
16. (*1.5) 编写函数**itos(int)**，它返回一个表示其**int**参数的**string**。

第21章 流

你看到的也就是所有你能得到的。

——Brian Kernighan

输入和输出——*ostream*——内部类型的输出——用户定义类型的输出——虚输出函数
——*istream*——内部类型的输入——无格式输入——流状态——用户定义类型的输入
——I/O异常——流的联结（*tying*）——哨位（*sentry*）——格式化整数和浮点数输出
——域和调整——操控符——标准操控符——用户定义操控符——文件流——关闭流——
字符串流——流缓存——现场（*locale*）——流的回调——*printf()*——忠告——练习

21.1 引言

为某种程序设计语言设计和实现一种通用的输入/输出功能是一项众所周知的困难工作。按照传统，I/O功能完全是为处理不多的几种内部数据类型设计的。然而，任何非平凡的C++都会使用许多用户定义类型，因此就必须处理这些类型的输入和输出问题。一种I/O机制应该是简单的、方便的、使用起来安全的、有效而灵活的，当然，最重要的还应该是完整的。没有人能够拿出一种使每个人都高兴的解决方案。因此，就应该使用户有可能提供另外的I/O机制，并有可能去扩充标准I/O功能，以适应特定应用的需要。

C++的设计就是要使用户可以定义新类型，使这些新类型的使用能与内部类型一样方便而有效。这就提出了一个很合理的要求：在C++里，应该只用每个程序员都能用的那些机制为C++提供一种I/O功能。这里要介绍的流I/O功能就是直面这种挑战的努力的成果。

21.2节 输出：被应用程序员看做输出的实际上是从各种类型（例如*int*、*char**和*Employee_record*等的对象）到字符序列的转换结果。这里将介绍一些机制，它们可用于描述内部类型和用户定义类型的输出。

21.3节 输入：介绍获取表示字符、串、其他内部类型和用户定义类型的输入的功能。

21.4节 格式化：人们常常会遇到对输出布局的特殊要求。例如，可能要求将*int*用十进制打印，指针用十六进制，浮点数必须以特定精度的形式出现。这里将要讨论格式化控制和提供它们所用的程序设计技术。

21.5节 文件和流：按默认约定，每个C++程序都能使用标准流，如标准输出（*cout*）、标准输入（*cin*）和错误输出（*cerr*）。为了使用其他设备或者文件，必须能创建流并将其附着到这些文件或设备上。本节将介绍打开和关闭文件，以及将流附着到文件和*string*上的那些机制。

21.6节 缓冲：为了提高I/O的效率，我们就必须使用缓冲技术，以适应数据的写（读）以及被写入（读出）的目标的情况。这里要介绍基本的缓冲流的技术。

21.7节 现场：一个*locale*（现场）是有关数值如何打印、哪些字符被当做字母等的一组

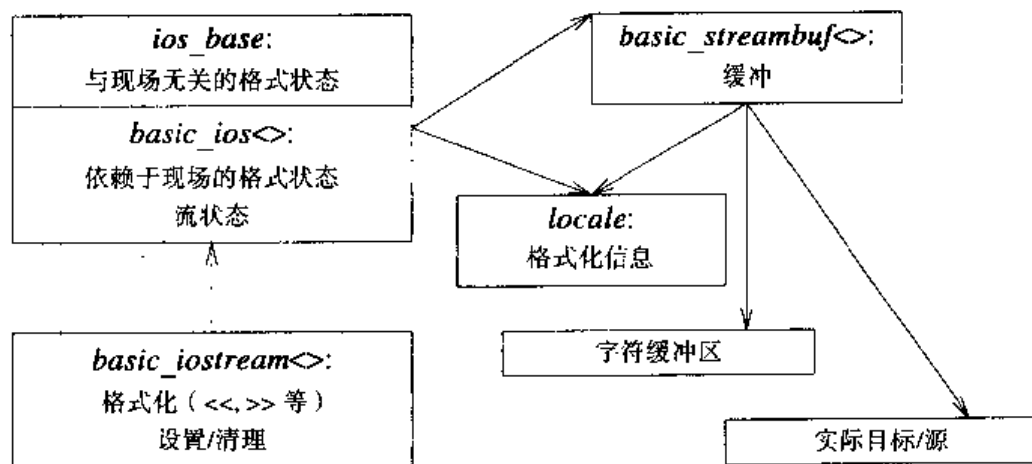
描述。它用于封装起许多文化差异。I/O系统只是隐式地使用现场，在这里只做简单的介绍。

21.8节 C的I/O：这里要讨论C `<stdio.h>` 库里的`printf()`函数以及C库与C++的`<iostream>`库之间的关系。

对于使用这个库而言，理解用于实现流库的各种技术并不是必须的。还有，在不同实现里所用的技术也可能不同。当然，实现I/O库是一项挑战性的工作，在实现中包含了许多技术实例，这些技术都可以应用到许多其他编程和设计工作中。

本章中有关流I/O系统的讨论将深入到使你能理解它的结构，将它用于最常见的I/O情况，以及为某些新的用户定义类型而扩充它。如果你需要实现标准流，提供新的流类型，或者提供一个新的现场，那么除了这里所展示的内容之外，你还需要一份语言标准、一本很好的系统手册，以及/或者许多工作代码的实例。

流I/O系统的关键组件可以用下面的图形表示：



从`basic_istream<>`出发的点箭头表明`basic_ios<>`是一个虚基类；实线箭头代表指针。用`<>`标记的类是模板，这些模板都用一个字符类型参数化的，并且包括一个`locale`。

流的概念和它所提供的一般性概念可以用于一大类通信问题。流已经被用于在不同的机器间传递对象（25.4.1节），用于加密消息流（21.10[22]），用于数据压缩和对象的持续性存储，还有大量其他工作。当然，这里的讨论将限制在简单的而向字符的输入和输出。

流I/O类和模板的声明（足够去引用它们，但并不将操作应用于它们）以及标准`typedef`由`<iosfwd>`给出。当你只想包含若干个而非全部I/O头文件时，偶尔会需要这个头文件。

21.2 输出

要想做到类型安全，并能统一地处理各种内部类型和用户定义类型，可以采用一个单一的重载函数名，并提供一组输出函数。例如，

```

put(cerr, "x = "); // cerr是错误输出流
put(cerr, x);
put(cerr, '\n');
  
```

根据参数类型确定对每个参数应调用哪一个`put`函数，这种解决方案已经用在一些语言里。但是，这样做重复太多。采用重载运算符`<<`表示“放入”是一种更好的记法形式，这就使程序员可以用一个语句输出一系列对象。例如，

```
cerr << "x = " << x << '\n';
```

如果 x 的值是123, 这个语句将向标准错误流 $cerr$ 打印出

```
x = 123
```

后跟一个换行符。类似地, 如果 x 的类型是`complex` (22.5节), 值为 $(1, 2.4)$, 该语句将在 $cerr$ 输出

```
x = (1, 2.4)
```

只要 x 属于某个定义了运算符`<<`的类型, 就可以采用这种风格。用户也很容易为一个新类型定义运算符`<<`。

为了避免那种使用输出函数的啰嗦方式, 就需要用一个输出运算符, 但为什么是`<<`呢? 发明一个新词法单词是不可能的 (11.2节)。赋值运算符曾经被作为输出和输出的候选, 但是大部分人看来希望对输入和输出采用不同的运算符。进一步说, `=`的约束关系也不对, 也就是说, `cout = a = b`的意思是`cout = (a = b)`而不是`(cout = a) = b` (6.2节)。我试过`<`和`>`运算符, 但“小于”和“大于”的意思在人们的心中如此牢固, 以致于从实践的观点上看, 新的输入输出语句很难阅读。

运算符`<<`和`>>`在内部类型中使用得不频繁, 因此不会引起问题。它们的对称形式可以用于提示“从”和“到”。在将它们用于I/O时, 我喜欢把`<<`说成是放入, 把`>>`说成是取出。偏爱更技术性的名字的人可能称它们为插入符和提取符。`<<`的优先级足够低, 这就使算术表达式可以直接作为运算对象, 而不必使用括号。例如,

```
cout << "a*b+c=" << a*b+c << '\n';
```

如果表达式里包含优先级比`<<`还低的运算符, 那么就必须使用括号了。例如,

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

在输出语句里同样可以包含左移运算符 (6.2.4节), 当然, 必须将它放进括号里:

```
cout << "a<<b=" << (a<<b) << '\n';
```

21.2.1 输出流

一个`ostream`是一种能将不同类型的值转换为字符序列的机制。通常, 这些字符随后通过低级的输出操作输出来。存在着许多种不同的字符 (20.2节), 它们的特征由`char_traits`描述 (20.2.1节)。因此, 一个`ostream`也就是通用的`basic_ostream`模板针对一种特定字符的专门化:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    virtual ~basic_ostream();
    // ...
};
```

这个模板及相关输出操作都在名字空间`std`里定义, 由`<ostream>`给出, 这个头文件里包含的是`<iostream>`中与输出有关的那一部分。

`basic_ostream`模板的参数控制着实现中所用的各种字符类型, 它们并不影响能够输出的值的类型。每个实现都直接支持针对常规的`char`实现的流和用宽字符实现的流:

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t> wostream;
```

在许多系统中，完全可能对通过 *wostream* 输出宽字符的流做深入的优化，使以字节作为输出单位的流根本无法与之相媲美。

也完全可能定义这样的流，使其中物理输出并不以字符的方式进行。当然，这种流已经超出了 C++ 标准的范畴，也超出了本书的范围（21.10[15]）。

basic_ios 基类在 `<ios>` 里给出。它控制着格式化（21.4节）、现场（21.7节）和对缓冲区的访问（21.6节）。它也定义了几个为了记述方便而用的类型：

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type; // 字符的整数值类型
    typedef typename Tr::pos_type pos_type; // 缓冲区中的位置
    typedef typename Tr::off_type off_type; // 缓冲区中的偏移量

    // ...另见21.3.3节、21.3.7节、21.4.4节、21.6.3节和21.7.1节 ...

    // 将赋值和复制构造函数作为私用（且不定义），以防复制（11.2.2节）
};
```

类 *basic_ios* 禁止复制构造和赋值（11.2.2节），这也意味着 *ostream* 和 *istream* 都不能复制。因此，如果你需要改变一个流的目标，你就必须去改变流缓冲区（21.6.4节），或者是间接地通过一个指针（6.1.7节）。

类 *ios_base* 包含着许多与所用字符类型无关的信息和操作。例如，用于浮点数输出的精度等。因为这种情况，它也不必是一个模板。

除了位于 *basic_ios* 里的那些 *typedef* 之外，流 I/O 库还用了一个表示缓冲区大小，以及在一个 I/O 操作中传递的字符数的有符号整数类型 *streamsize*。类似地，它还提供了一个称为 *streamoff* 的 *typedef*，用于表示流和缓冲区里的偏移量。

在 `<iostream>` 里声明了几个标准流：

```
ostream cout; // char 的标准输出流
ostream cerr; // 用于错误信息的标准非缓冲输出流
ostream clog; // 用于错误信息的标准输出流

wostream wcout; // 对应于 cout 的宽字符流
wostream wcerr; // 对应于 cerr 的宽字符流
wostream wclog; // 对应于 clog 的宽字符流
```

cerr 和 *clog* 流写的目标相同，其简单差别就在于是否为输出提供缓冲。*cout* 写的目标与 *stdout* 一样（21.8节），而 *cerr* 和 *clog* 写的目标都与 *stderr* 一样。程序员可以根据需要创建更多的流（21.5节）。

21.2.2 内部类型的输出

类 *ostream* 还定义了处理内部类型输出的 `<<`（“放入”）运算符：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    basic_ostream& operator<<(short n);
```

```

    basic_ostream& operator<< (int n);
    basic_ostream& operator<< (long n);

    basic_ostream& operator<< (unsigned short n);
    basic_ostream& operator<< (unsigned int n);
    basic_ostream& operator<< (unsigned long n);

    basic_ostream& operator<< (float f);
    basic_ostream& operator<< (double f);
    basic_ostream& operator<< (long double f);

    basic_ostream& operator<< (bool n);
    basic_ostream& operator<< (const void* p);           // 写指针值

    basic_ostream& put (Ch c);           // 写出c
    basic_ostream& write (const Ch* p, streamsize n);   // p[0]..p[n-1]

    // ...
};

```

`put()` 和 `write()` 函数简单地写出字符。这样，输出字符的 `<<` 就不必是成员了。以一个字符为参数的 `operator<<()` 函数可以通过 `put()` 实现为非成员函数：

```

template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, Ch);
template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, char);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, char);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, signed char);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, unsigned char);

```

与此类似，还为写出0结尾的字符数组提供了 `<<` 的定义：

```

template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const Ch*);
template<class Ch, class Tr>
    basic_ostream<Ch, Tr>& operator<< (basic_ostream<Ch, Tr>&, const char*);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const char*);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const signed char*);
template<class Tr>
    basic_ostream<char, Tr>& operator<< (basic_ostream<char, Tr>&, const unsigned char*);

```

对 `string` 的输出运算符在 `<string>` 里给出，见20.3.15节。

`operator<<()` 返回调用所针对的那个 `ostream` 的引用，这就使另一个 `operator<<()` 可以应用到这个结果上。例如，

```
cerr << "x = " << x;
```

假设这里的 `x` 是个 `int`，整个语句将被解释为

```
operator<< (cerr, "x = ") . operator<< (x);
```

特别地，这就意味着可以在一个输出语句里打印多个数据项，它们将按照预期的顺序打印出来，从左到右。例如，

```

void val(char c)
{
    cout << "int(' " << c << " ') = " << int(c) << '\n';
}

int main()
{
    val('A');
    val('Z');
}

```

在一个采用ASCII的实现中, 这将打印出

```

int('A') = 65
int('Z') = 90

```

注意, 字符文字量的类型是`char` (4.3.1节), 所以`cout << 'Z'` 将打印出字母`Z`, 而不是整数值`90`。

按照默认约定, `bool`值将打印出`0`或`1`。如果不喜欢这个情况, 你就可以从 `<iomanip>` 里设置格式化标志`boolalpha` (21.4.6.2节), 从而得到`true`或`false`。例如,

```

int main()
{
    cout << true << ' ' << false << '\n';
    cout << boolalpha; // 用符号形式表示真和假
    cout << true << ' ' << false << '\n';
}

```

将打印出

```

1 0
true false

```

说的更准确些, `boolalpha`保证我们可以得到一个`bool`值的与现场有关的表示形式。通过适当设置我所用的现场 (21.7节), 我也可以得到:

```

1 0
sant falsk

```

格式化浮点数, 整数使用的基数等将在21.4节讨论。

函数`ostream::operator<<(const void*)` 按照适合所用机器的系统结构的某种形式打印出一个指针值。例如,

```

int main()
{
    int* p = new int;
    cout << "local " << &p << ", free store " << p << '\n';
}

```

在我的机器上打印出

```

local 0x7fffead0, free store 0x500c

```

其他系统可能对打印指针值采用与此不同的规定。

21.2.3 用户定义类型的输出

考虑用户定义类型`complex` (11.3):

```

class complex {
public:

```

```

double real() const { return re; }
double imag() const { return im; }
// ...
};

```

可以按如下方式为新类型`complex`定义 `<<` 运算符:

```

ostream& operator<<(ostream&s, const complex&z)
{
    return s<< '('<<z.real()<< ','<<z.imag()<< ')';
}

```

这个 `<<` 的使用与对于内部类型的 `<<` 完全一样。例如,

```

int main()
{
    complex x(1,2);
    cout<<"x="<<x<<'\n';
}

```

产生

```
x = (1, 2)
```

为用户定义类型定义输出运算符不需要修改`ostream`类的声明,这当然是很幸运的,因为`ostream`在`<ostream>`里定义、它是用户不能也不应该去修改的。不允许向`ostream`里添加功能也是一种保护,以防无意中破坏那里的数据结构,也使我们有可能修改`ostream`的实现,同时又不会影响用户程序。

21.2.3.1 虚输出函数

`ostream`的成员函数都不是`virtual`。程序员能增加的输出操作不可能作为成员函数,因此它们也不可能是`virtual`。这样做有一个原因,就是希望使简单操作具有接近最优的性能,例如将单个字符放入缓冲区的操作。这正是影响运行时间最关键的地方,也是必须做在线处理的地方。虚函数只用在处理缓冲区上溢或者下溢的操作,以取得灵活性(21.6.4节)。

然而,程序员有时也会希望输出一个只知道其基类的对象。因为不知道确切的类型,通过简单地为新类型定义一个 `<<` 就无法得到正确的输出。这时就需要换一种方式,在抽象基类里提供一个虚输出函数:

```

class My_base {
public:
    // ...

    virtual ostream& put(ostream&s) const = 0;    // 将 *this写入s
};

ostream& operator<<(ostream&s, const My_base&r)
{
    return r.put(s);    // 用正确的put()
}

```

也就是说,在这里`put()`是一个虚函数,它保证在 `<<` 中能使用正确的输出操作。

有了上面的东西,我们就可以写

```

class Sometype : public My_base {
public:
    // ...

    ostream& put(ostream&s) const;    // 实际输出函数:覆盖My_base::put()
}

```

```
};

void f(const My_base& r, Sometype& s)    // 用 <<, 它能调用正确的put()
{
    cout << r << s;
}
```

这就把虚`put()`集成到了由`ostream`和`<<`提供的框架中。这种技术是通用性的, 在提供类似虚函数的操作时非常有用, 但是, 这里需要基于第二个参数在运行时做出选择。

21.3 输入

对输入的处理方式与输出类似, 用类`istream`为一小组标准类型提供输入运算符`>>` (“取出”)。对用户定义类型也可以定义`operator>>()`。

21.3.1 输入流

在`<istream>`里包含着`<iostream>`中与输入有关的部分。与`basic_ostream` (21.2.1节)平行, 在`<istream>`定义了`basic_istream`, 它具有下面的样子:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    virtual ~basic_istream();

    // ...
};
```

基类`basic_ios`已经在21.2.1节介绍过了。

`<istream>`中提供了两个标准输入流`cin`和`wcin`:

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

istream cin;    // char的标准输入流
wistream wcin; // wchar_t的标准输入流
```

`cin`流读入的来源与C的`stdin` (21.8节)相同。

21.3.2 内部类型的输入

`istream`为内部类型提供了`>>`运算符:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    // 格式化的输入;

    basic_istream& operator>>(short& n);           // 读入n
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(long& n);

    basic_istream& operator>>(unsigned short& u); // 读入u
    basic_istream& operator>>(unsigned int& u);
    basic_istream& operator>>(unsigned long& u);

    basic_istream& operator>>(float& f);           // 读入f
```



```

    basic_istream& operator>>(double& f);
    basic_istream& operator>>(long double& f);

    basic_istream& operator>>(bool& b);           // 读入b
    basic_istream& operator>>(void*& p);          // 将指针值读入p

    // ...
};

```

输入函数`operator>>()`的定义都采用如下风格:

```

istream& istream::operator>>(T& tvar)    // T是一个声明了istream::operator>>的类型
{
    // 跳过空白, 将一个T读入tvar
    return *this;
}

```

由于`>>`跳过空白, 所以你可按如下方式读入一个由空白分隔的整数序列:

```

int read_ints(vector<int>& v)    // 填充v, 返回读入整数的个数
{
    int i = 0;
    while (i < v.size() && cin >> v[i]) i++;
    return i;
}

```

输入中的非`int`将导致输入操作失败并结束输入循环。例如, 输入

`1 2 3 4 5.6 7 8`

将使`read_ints()`读入

`1 2 3 4 5`

这5个整数, 并将圆点留下, 使之成为供输入读的下一个字符。空白定义为标准的C空白(空格、制表符、换行符、换页符和回车符), 通过调用`<cctype>`里定义的`isspace()`确定(20.4.2节)。

在使用`istream`时, 最容易犯的错误就是没注意到输入并没有按预期的方式出现, 原因是实际输入并不具有所需的格式。我们应该在使用可能输入的值之前先检查输入流的状态(21.3.3节), 或者使用异常(21.3.6节)。

输入所需的格式在当前的现场里描述(21.7节)。按照默认约定, `bool`值`true`和`false`分别用`1`和`0`表示。整数必须是十进制的, 浮点数具有它们在C++程序里书写时所具有的形式。通过设置`basefield`(21.4.2节), 可以将`0123`作为八进制形式读入取得十进制值`83`, 也可以将`0xff`作为十六进制形式读入取得十进制值`255`。用于读入指针的格式则完全依赖于实现(请看一下你所用的实现怎样完成这些事情)。

令人惊疑的是, 在这里没有读入字符的`>>`成员。原因是字符的`>>`可以通过字符输入操作`get()`实现(21.3.4节), 所以它就不必作为成员了。我们可以读字符到流本身的字符类型中。如果该字符类型是`char`, 我们也可以读入到`signed char`和`unsigned char`:

```

template<class Ch, class Tr>
basic_istream<Ch, Tr>& operator>>(basic_istream<Ch, Tr>&, Ch&);

template<class Tr>
basic_istream<char, Tr>& operator>>(basic_istream<char, Tr>&, unsigned char&);

template<class Tr>
basic_istream<char, Tr>& operator>>(basic_istream<char, Tr>&, signed char&);

```

从用户的角度看, >> 是不是成员根本就无所谓。

像其他 >> 运算符一样, 这些函数也先跳过空白。例如,

```
void f()
{
    char c;
    cin >> c;
    // ...
}
```

这就将来自 *cin* 的第一个非空白字符放进了 *c*。

此外, 我们还可以读入到字符数组里:

```
template<class Ch, class Tr>
    basic_istream<Ch, Tr>& operator>> (basic_istream<Ch, Tr>&, Ch*);
template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, unsigned char*);
template<class Tr>
    basic_istream<char, Tr>& operator>> (basic_istream<char, Tr>&, signed char*);
```

这些操作都先跳过空白, 而后不断将字符读入它们的数组参数里, 直到又遇到空白或者文件结束符。它们最后用一个 *0* 结束这个字符串。很清楚, 这样做存在着产生溢出的极大危险性, 所以, 通常更好的选择是读入到一个 *string* 里 (20.3.15 节)。当然, 你可以描述 >> 的最大读入字符数, 用 *is.width(n)* 确定, 它指定下一次对 *is* 的 >> 最多将 *n-1* 个字符读入到数组里。例如,

```
void g()
{
    char v[4];
    cin.width(4);
    cin >> v;
    cout << "v = " << v << endl;
}
```

这将至多读入3个字符到 *v*, 并加上结束符 *0*。

对 *istream* 设置 *width()* 只影响紧随其后的一次对数组的 >>, 不会对其他变量类型的读入产生任何影响。

21.3.3 流状态

每个流 (*istream* 或 *ostream*) 都有一个与之相关联的状态。出错和非标准条件都通过适当地设置和检测这个状态来处理。

在 *basic_istream* 的基类 *basic_ios* 里 (在 <ios>), 可以找到流的各种状态:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    bool good() const;    // 下一个操作可能成功
    bool eof() const;     // 遇到文件结束
    bool fail() const;    // 下一个操作将失败
    bool bad() const;     // 流已破坏

    iostate rdstate() const;    // 取io状态标志
    void clear(iostate f = goodbit); // 设置io流状态
```

```

void setstate(iostate f) { clear(rdstate() | f); } // 将f加进io状态标志

operator void* () const;           // 如果 !fail() 则非零
bool operator! () const { return fail(); }

// ...
};

```

如果状态是`good()`，则说明前一个操作成功，下一个输入操作可能成功；否则它就一定失败。仅就接收读入的变量而论，将操作应用于状态不是`good()`的流将不产生任何作用。如果我们试图读入到一个变量`v`而操作失败，`v`的值应该没有改变（如果`v`是某个类型的，而该类型由`istream`或者`ostream`的成员函数处理，那么`v`就不会改变）。在`fail()`和`bad()`之间的差异相当微妙。如果状态是`fail()`但并不同时是`bad()`，那么就可以假设流并未破坏，也没有字符丢失。如果状态是`bad()`，那就一切都完了。

一个流的状态用一组标志表示。就像用于表示流行为的大部分常量一样，这些标志也都在`basic_ios`的基类`ios_base`里定义：

```

class ios_base {
public:
    // ...

    typedef implementation_defined2 iostate;
    static const iostate badbit,      // 流已破坏
                      eofbit,        // 遇到文件结束
                      failbit,       // 下一个操作将失败
                      goodbit;       // goodbit == 0

    // ...
};

```

I/O状态标志可以直接操作。例如，

```

void f()
{
    ios_base::iostate s = cin.rdstate(); // 返回一组iostate位
    if (s & ios_base::badbit) {
        // cin的字符可能丢失
    }
    // ...
    cin.setstate(ios_base::failbit);
    // ...
}

```

如果将一个流当做条件使用，这个流的状态就会通过`operator void*()`或者`operator!()`检测。如果`!fail()`，这个检查就成功，否则就失败。例如，可以用如下方式写出一个通用的复制函数：

```

template<class T> void iocopy(istream& is, ostream& os)
{
    T buf;
    while (is >> buf) os << buf << '\n';
}

```

这个`is >> buf`返回对`is`的引用，该引用将被对`istream::operator void*()`的调用检测。例如，

```

void f(istream& i1, istream& i2, istream& i3, istream& i4)
{

```

```

    iocopy<complex>(i1, cout);    // 复制复数
    iocopy<double>(i2, cout);     // 复制double
    iocopy<char>(i3, cout);       // 复制char
    iocopy<string>(i4, cout);     // 复制空白分隔的单词
}

```

21.3.4 字符的输入

运算符 `>>` 的目的是做格式化的输入，也就是说，用于读入具有某种预期类型和格式的对象。如果我们不想这样做，而希望按字符的方式读入一些字符，而后再检查它们，那么就應該用 `get()` 函数：

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    // 非格式化输入：

    streamsize gcount() const;    // 由上一次get()读入的字符个数

    int_type get();               // 读一个Ch (或Tr::eof())

    basic_istream& get(Ch& c);    // 读一个Ch到c

    basic_istream& get(Ch* p, streamsize n);    // 以换行符作为结束符
    basic_istream& get(Ch* p, streamsize n, Ch term);

    basic_istream& getline(Ch* p, streamsize n);    // 以换行符作为结束符
    basic_istream& getline(Ch* p, streamsize n, Ch term);

    basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof());
    basic_istream& read(Ch* p, streamsize n);    // 读入至多n个字符
    // ...
};

```

此外，`<string>` 也为标准 `string` 提供了 `getline()` (20.3.15节)。

`get()` 和 `getline()` 函数对空白字符的处理与对其他字符完全一样。这些函数只是用于输入操作，对于输入的那些字符的意义并没有任何假定。

函数 `istream::get(char&)` 将一个字符读进它的参数里。举例来说，一个一个地复制字符的程序写出来是下面的样子：

```

int main()
{
    char c;
    while(cin.get(c)) cout.put(c);
}

```

三个参数的 `s.get(p, n, term)` 将至多 $n - 1$ 个字符读入 `p[0] .. p[n - 2]`。对这种 `get()` 的调用将总在它放入缓冲区的那些字符 (如果有) 最后放一个 `0`。所以，`p` 应该指向一个至少有 `n` 个字符的数组。第三个参数 `term` 给定一个结束符。这种三个参数的 `get()` 的典型使用是将一个“行”读入一个固定大小的缓冲区供后面使用。例如，

```

void f()
{
    char buf[100];
    cin >> buf;    // 可疑：早晚会溢出
    cin.get(buf, 100, '\n');    // 安全
}

```

```

    // ...
}

```

如果输入中遇到结束符，这个字符将作为第一个未读的字符留在流中。绝不要两次调用`get()`而忘记删除结束符。例如，

```

void subtle_infinite_loop()
{
    char buf[256];
    while (cin) {
        cin.get(buf, 256);    // 读一行
        cout << buf;         // 打印一行。呜呼：忘记从cin删除 '\n'
    }
}

```

这种例子是说明应该更多地用`getline()`而不是`get()`的一个好理由。`getline()`的行为正好与`get()`相对应，只是它将把结束符从`istream`清除。例如，

```

void f()
{
    char word[MAX_WORD][MAX_LINE];    // MAX_WORD数组，每个MAX_LINE个字符
    int i = 0;
    while (cin.getline(word[i++], MAX_LINE, '\n') && i < MAX_WORD) {
        // ...
    }
}

```

如果效率不那么重要，那么最好还是读入到`string`（3.6节、20.3.15节）里，那样做就不会出现最常见的分配和溢出问题。然而，在实现那一类高级功能时，就需要利用`get()`、`getline()`和`read()`。这些相对比较难弄的界面就是我们要付出的代价，为了速度，为了不必重新扫描输入去找出输入操作的结束位置，为了能可靠地限制输入的字符个数，等等。

调用`read(p, n)`将最多`n`个字符读入`p[0] .. p[n - 1]`。这个读入函数不依赖于结束符，也不在其目标的最后放结束符`0`。因此它能实际读入`n`个字符（而不是`n - 1`个）。换句话说，它就是简单地读入字符，而不试图将其目标做成C风格字符串。

`ignore()`函数像`read()`一样读字符，但它并不将读入的字符存到任何地方。它也像`read()`一样能实际读`n`个字符（而不是`n - 1`个）。由`ignore()`读入字符的个数默认为1，所以，不用参数调用`ignore()`就意味着“丢掉下一个字符”。`ignore()`也可以用一个结束符（像`getline()`一样），在读入中遇到这个结束符时，也将它从输入流中删除。注意，`ignore()`的默认结束符是文件结束。

对于所有这些函数而言，究竟什么导致了输入结束都不是很明确的，即使记住了有关的结束准则，要确定究竟是什么情况导致结束也可能还是很困难。当然，我们总可以询问是否遇到了文件结束（21.3.3节）。还有，`gcount()`能给出最近一次非格式化输入函数的调用从流中读入的字符个数。例如，

```

void read_a_line(int max)
{
    // ...
    if (cin.fail()) {        // 呜呼：错误输入格式
        cin.clear();         // 清除输入标志（21.3.3节）
        cin.ignore(max, '\n'); // 跳过分号
    }
    if (!cin) {

```

```

        // 呜呼：遇到流结束
    }
    else if (cin.gcount() == max) {
        // 呜呼：读入了最大数目的字符
    }
    else {
        // 发现并丢弃的是分号
    }
}
}

```

不幸的是，在读入达到最大字符个数时，就没有办法知道是否遇到了结束符（它是否正好是最后一个字符）。

无参数的`get()`是`<cstdio>`中`getchar()`的`<iostream>`版本（21.8节）。它简单地读入一个字符，返回该字符的数值。在这样做时，`get()`不对字符的种类做任何假设。如果不存在能返回的字符，`get()`将返回合适的“文件结束”标志（即该流的`traits_type::eof()`），并将这个`istream`设置为`eof`状态（21.3.3节）。例如，

```

void f(unsigned char* p)
{
    int i;
    while ((i = cin.get()) && i != EOF) {
        *p++ = i;
        // ...
    }
}

```

`EOF`是在普通`char`的`char_traits`里`eof()`的值。`EOF`在`<iostream>`里给出。这样，这个循环原本也可以写成`read(p, MAX_INT)`，但这里假定我们想写显式循环，是因为要在读入过程中查看每个字符。人们都说，C最强之处在于它能读入一个字符并决定对它什么也不做，而且可以快速地完成这件事情。这确实是一种非常重要的、常被低估的能力，也是C++希望保留的一种功能。

标准头文件`<cctype>`定义了一些函数，在处理输入时非常有用（20.4.2节）。例如，能从流中读入空白字符的`eatwhite()`函数可以定义如下：

```

istream& eatwhite(istream& is)
{
    char c;
    while (is.get(c)) {
        if (!isspace(c)) { // c是空白字符吗？
            is.putback(c); // 将c放回输入缓冲区
            break;
        }
    }
    return is;
}

```

调用`is.putback(c)`将使`c`成为下一次对流`is`读入时得到的字符（21.6.4节）。

21.3.5 用户定义类型的输入

同样可以为用户定义类型定义输入函数，其方式与定义输出函数完全一样。但是，输入函数要求其第二个参数必须是一个非`const`引用。例如，

```

istream& operator>>(istream& s, complex& a)
/*
    对复数的输入格式 (f表示浮点数):
        f
        (f)
        (f,f)
*/
{
    double re = 0, im = 0;
    char c = 0;

    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(ios_base::failbit); // 设置状态
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re, im);
    return s;
}

```

尽管错误处理代码不多，但这样实际上已经能处理大部分错误了。局部变量`c`先做初始化，以防它的值在第一个`>>`操作失败后正好很偶然地是`'('`。最后对流状态的检查保证只在一切都完好无误时才去改变`a`的值。如果发现一个格式错，这个流的状态将被设置为`failbit`。没有将流设置为`badbit`是因为它本身并没有破坏。用户可以重置流（用`clear()`），并可能跳过导致问题的一些字符，继续从流中提取有用信息。

这里设置流状态的操作被称做`clear()`，因为它最常见的用途就是将流的状态重置为`good()`，`ios_base::goodbit`是`clear()`的默认参数值（21.3.3节）。

21.3.6 异常

在每个I/O操作后都检查错误确实很不方便，所以，对于出错，最常见的问题就是在必须的地方却没有处理。特别是人们一般都不检查输出操作的错误，虽然它们有时也会失败。

`clear()`是仅有的能够直接修改流状态的函数。这样，使流状态改变能被注意到的一种明显方式就是让`clear()`去抛出一个异常。`basic_ios`的成员函数`exceptions()`做的就是这件事：

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...

    class failure; // 异常类（见14.10节）

    iostate exceptions() const; // 取出异常状态
    void exceptions(iostate except); // 设置异常状态

    // ...
};

```

例如，

```
cout.exceptions( ios_base::badbit | ios_base::failbit | ios_base::eofbit );
```

将要求`clear()`在`cout`进入状态`bad`、`fail`或`eof`时(也就是说,在`cout`上的任何输出操作未能毫无差错地完成时)抛出一个`ios_base::failure`异常。如果必要,我们也可以检查`cout`,去弄清到底是什么出了错。类似地,

```
cin.exceptions( ios_base::badbit | ios_base::failbit );
```

使我们能捕捉到一种比较常见的情况,在这种情况下输入的格式并非如我们的预期,所以这时对于流的输入操作无法返回一个值。

不带参数去调用`exceptions()`将返回触发异常的I/O状态标志集。例如,

```
void print_exceptions( ios_base& ios )
{
    ios_base::iostate s = ios.exceptions();
    if (s & ios_base::badbit) cout << "throws for bad";
    if (s & ios_base::failbit) cout << "throws for fail";
    if (s & ios_base::eofbit) cout << "throws for eof";
    if (s == 0) cout << "doesn't throw";
}
```

I/O异常的基本用途是捕捉未必会出现的(因而也常常被忽略的)错误。另一种用途是控制I/O。例如,

```
void readints( vector<int>& s )           // 并不是我喜欢的风格
{
    ios_base::iostate old_state = cin.exceptions(); // 保存异常状态
    cin.exceptions( ios_base::eofbit );           // 遇到eof抛出

    for ( ;; )
        try {
            int i;
            cin >> i;
            s.push_back(i);
        }
        catch( ios_base::failure ) {
            // 可以: 遇到文件结束
            break;
        }

    cin.exceptions( old_state );           // 恢复异常状态
}
```

对异常的这种使用方式,要问的问题是:“这是一个错误吗?”或者“实际存在异常吗?”(14.5节)。我常常发现对两个问题的回答都是“不”。因此我更喜欢直接去处理流状态。对于在函数里能用局部控制结构处理的问题,采用异常也很少能做得更好。

21.3.7 流的联结

`basic_ios`函数`tie()`用于设置或解开在一个`istream`和一个`ostream`之间的联结:

```
template <class Ch, class Tr = char_traits<Ch> >
class std::basic_ios : public ios_base {
    // ...

    basic_ostream<Ch, Tr>* tie() const;           // 取得指向联结的流的指针
    basic_ostream<Ch, Tr>* tie( basic_ostream<Ch, Tr>* s ); // 将 *this与s联结

    // ...
};
```


考虑

```
string get_passwd()
{
    string s;
    cout << "Password: ";
    cin >> s;
    // ...
}
```

我们怎么能保证“Password:”能在读入操作执行之前出现在屏幕上？对于`cout`的输出有缓冲，所以，如果`cin`和`cout`相互独立，在输出缓冲区满之前，“Password:”将不会出现在屏幕上。问题的答案是：通过操作`cin.tie(&cout)`将`cout`联结于`cin`。

当一个`ostream`被联结于一个`istream`之后，每当在这个`istream`上的输入操作导致下溢时（也就是说，当需要从最终输入源获取新字符，以完成一个输入操作时），与之联结的`ostream`的缓冲区都将被刷新。这样，

```
cout << "Password: ";
cin >> s;
```

就等价于

```
cout << "Password: ";
cout.flush();
cin >> s;
```

一个流同时至多只能有一个联结于其上的`ostream`。调用`s.tie(0)`将使`s`与它原来与之联结的那个流脱开（如果存在的话）。与其他许多设置值的流函数一样，`tie(s)`返回原来的联结流或者0。以无参数方式调用`tie()`将返回当前的值，且不改变联结关系。

对于标准流，`cout`被联结于`cin`，`wcout`被联结于`wcin`。`cerr`流不必联结，因为它是不缓冲的，而`clog`流不是为用户交互提供的功能。

21.3.8 哨位

在我为`complex`写`<<`和`>>`运算符时，我并不担心联结的流（21.3.7节），也不担心流状态的改变会导致异常（21.3.6节）。我假定（而且是正确的）库里提供的函数能帮我关照这些事情。但情况到底怎么样？在这里有数十个函数，如果我们不得不写出极其复杂代码去处理联结的流、`locale`（21.7节）、异常等，那么代码就会变得一团糟。

这里要采用的方式就是通过一个`sentry`类提供一些公共的代码，通过`sentry`的构造函数去提供需要先行执行的代码（或称“前缀代码”）（例如，刷新联结的流）；通过`sentry`的析构函数去提供需要在最后执行的代码（或称“后缀代码”）（例如，由状态改变引起抛出异常）：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
    // ...
    class sentry;
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream<Ch, Tr>::sentry {
public:
    explicit sentry(basic_ostream<Ch, Tr>& s);
```

```

    ~sentry();
    operator bool();

    // ...
};

```

这样就把有关的公共代码提取出来了，现在单独的函数就可以写成下面样子：

```

template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& basic_ostream<Ch, Tr>::operator<<(int i)
{
    sentry s(*this);
    if (!s) { // 检查是否一切正常，可以开始输出
        setstate(failbit);
        return *this;
    }

    // 输出int
    return *this;
}

```

这种通过一个类，采用构造函数和析构函数提供公共的前缀和后缀代码的技术在许多环境中都非常有用。

很自然，*basic_istream*有一个类似的*sentry*成员类。

21.4 格式化

21.2节里的例子都是所谓的非格式化输出，也就是说，都是按照默认规则将对象转为字符序列；但程序员常常需要更细节的控制。举例说，我们可能需要控制一个输出操作所用的空格数或者数值输出的格式。与此类似，有时也需要显式地去控制输入的某些方面。

对I/O格式化的控制位于类*basic_ios*及其基类*ios_base*里面。例如，在类*basic_ios*里保存着在读写整数时所用的基数（十进制、八进制或者十六进制）、读写浮点数的精度等。它还包含了一些函数，用于设置和检查各个流里的这些控制变量。

类*basic_ios*是*basic_istream*和*basic_ostream*的基类，所以，格式控制是基于各个独立的流进行的。

21.4.1 格式状态

I/O的格式化由*ios_base*里为流定义的一组标志和整数值控制：

```

class ios_base {
public:
    // ...
    // 格式标志名：

    typedef implementation_defined1 fmtflags;
    static const fmtflags
        skipws,           // 输入时跳过空白
        left,             // 域调整：值后填充
        right,            // 值前填充
        internal,         // 在符号和价值之间填充
        boolalpha,        // 用符号形式表示真假
        dec,              // 整数的基数：以10为基数输出（十进制）

```

```

    hex,                // 以16为基数输出（十六进制）
    oct,                // 以8为基数输出（八进制）

    scientific,         // 浮点数格式: d.dddddEdd
    fixed,              // 浮点数格式: dddd.dd

    showbase,           // 输出前缀，八进制加0，十六进制加0x
    showpoint,          // 打印尾随的0
    showpos,            // 正整数加显式的 + 符号
    uppercase,          // 用E和X而不是e和x

    adjustfield,         // 与域调整有关的标志组（21.4.5节）
    basefield,           // 与整数基数有关的标志组（21.4.2节）
    floatfield,          // 与浮点数输出有关的标志组（21.4.3节）

    unitbuf;            // 每次输出操作之后刷新

    fmtflags flags() const;    // 读标志
    fmtflags flags(fmtflags f); // 设置标志

    fmtflags setf(fmtflags f) { return flags(flags() | f); } // 添加标志
    // 在掩码中清除及设置标志:
    fmtflags setf(fmtflags f, fmtflags mask) { return flags((flags() & ~mask) | (f & mask)); }
    void unsetf(fmtflags mask) { flags(flags() & ~mask); } // 清除标志

    // ...
};

```

这些标志的值由实现定义。使用时应该全部用符号名，绝不要用数值，即使某些值在你所用的实现上现在正好是正确的。

将界面定义为一组标志，并提供设置和清除这些标志的操作，这是一种经过时间考验的或者说是陈旧的技术了，其主要优点是使用户可以组合起一组选项。例如，

```
const ios_base::fmtflags my_opt = ios_base::left | ios_base::oct | ios_base::fixed;
```

这就使我们可以来回传递一些选项，在需要时设置它们。例如，

```

void your_function(ios_base::fmtflags opt)
{
    ios_base::fmtflags old_options = cout.flags(opt); // 在old_options中保存原选项，设置新选项
    // ...
    cout.flags(old_options); // 恢复原来的选项
}

void my_function()
{
    your_function(my_opt);
    // ...
}

```

函数`flags()`返回原来的选项集。

可以读取或者设置所有选项，这就使我们可以设置单独的标志。例如，

```
myostream.flags(myostream.flags() | ios_base::showpos);
```

这使`myostream`在显示出的每个正数前面加一个显式的 + 符号，且不影响其他选项。这里先读出原来的选项集，用“或”操作将`showpos`加进集合。函数`setf()`做的就是这件事，所以，上面例子可以等价地写成

```
myostream.setf(ios_base::showpos);
```

一旦设置后，该标志就将一直保存这个值，直到它被清除。

通过显式设置和清除标志的方式控制I/O选项既显得生硬也很容易出错。对于那些比较简单的情況，操控符（21.4.6节）提供了一个更清晰的界面。用标志位的方式控制流状态，最好是作为一种实现技术来研究，而不是在界面设计时使用。

21.4.1.1 复制格式状态

用*copyfmt()*可以复制一个流的完整格式状态：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    basic_ios& copyfmt(const basic_ios& f);
    // ...
};
```

*copyfmt()*并不复制流的缓冲区（21.6节）和缓冲区的状态，但是它复制状态中其余的东西，包括所要求的异常（21.3.6节）和用户给状态添加的所有东西（21.7.1节）。

21.4.2 整数输出

将一个新选项“或”到*flags()*上，或者采用*setf()*的方式，都只能用于由一个二进制位控制的属性，打印整数所用的基数或者浮点数输出格式都不属于这种情况。对于这些选项，需要描述的风格可能无法用一个位或者一组相互独立的位描述。

*<iostream>*采取的解决方法是提供一个*setf()*版本，它有第二个“伪参数”，用于指明除了新值之外我们还希望设置哪些选项。例如，

```
cout.setf(ios_base::oct, ios_base::basefield); // 八进制
cout.setf(ios_base::dec, ios_base::basefield); // 十进制
cout.setf(ios_base::hex, ios_base::basefield); // 十六进制
```

设置整数的基数而对流状态的其他方面都没有影响。一旦设置后，输出就将一直采用这个基数，直到重新设置为止。例如，

```
cout << 1234 << ' ' << 1234 << ' '; // 默认：十进制
cout.setf(ios_base::oct, ios_base::basefield); // 八进制
cout << 1234 << ' ' << 1234 << ' ';

cout.setf(ios_base::hex, ios_base::basefield); // 十六进制
cout << 1234 << ' ' << 1234 << ' ';
```

将产生1234 1234 02322 02322 4d2 4d2。

如果我们需要表现出每个数字所用的基数，那么就可以设置*showbase*。这样，在前面操作之前增加

```
cout.setf(ios_base::showbase);
```

我们将得到1234 1234 02322 02322 0x4d2 0x4d2。标准操控符（21.4.6.2节）提供了一种描述整数输出基数的更优美的方式。

21.4.3 浮点数输出

浮点数的输出通过格式和精度控制：

- 一般格式 (general) 让具体实现去为输出值选一种表现格式, 使之能在可用空间内以最佳方式维持这个值。精度刻画了数字的最大位数。这种格式对应于 `printf()` 的 `%g` (21.8节)。
- 科学格式 (scientific) 通常用小数点前一位数字以及一个指数部分的方式表现数值。精度刻画的是小数点之后的最大位数。它对应于 `printf()` 的 `%e` (21.8节)。
- 定点格式 (fixed) 将值表示为一个整数部分, 后跟小数点及一个小数部分。精度刻画的是小数点之后的最大数字位数。它对应于 `printf()` 的 `%f` (21.8节)。

我们通过状态操控函数来控制浮点数输出的格式。特别是, 我们可以设置浮点数值值的记法形式, 而不会对流状态的其他方面造成影响。例如,

```
cout << "default:\n" << 1234.56789 << '\n';

cout.setf(ios_base::scientific, ios_base::floatfield); // 采用科学格式
cout << "scientific:\n" << 1234.56789 << '\n';

cout.setf(ios_base::fixed, ios_base::floatfield);      // 采用定点格式
cout << "fixed:\n" << 1234.56789 << '\n';

cout.setf(ios_base::fmtflags(0), ios_base::floatfield); // 恢复默认格式 (即一般格式)
cout << "default:\n" << 1234.56789 << '\n';
```

将产生

```
default: 1234.57
scientific: 1.234568e+03
fixed: 1234.567890
default: 1234.57
```

默认的精度是6 (对于所有格式)。精度由 `ios_base` 的一个成员函数控制:

```
class ios_base {
public:
    // ...
    streamsize precision() const;          // 取得精度
    streamsize precision(streamsize n);    // 设置精度 (并取得原有精度)
    // ...
};
```

对 `precision()` 的调用将影响这个流的所有浮点数I/O操作, 直到下一次调用 `precision()` 为止。这样,

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

将产生

```
1234.5679 1234.5679 123456
1235 1235 123456
```

注意, 对浮点数做的是舍入而不是简单截断, `precision()` 并不影响整数输出。

`uppercase` 标志 (21.4.1节) 确定按科学格式时是用 `e` 还是用 `E` 去表示指数。

操控符提供了一种更优美的描述浮点数输出格式的方式 (21.4.6.2节)。

21.4.4 输出域

我们常常需要用正文填满一个输出行里的一段特定空区, 希望用正好 `n` 个而不是更少一些

的字符（而且只在正文不够时才做填充）。为此，我们要指定域的宽度，并指定需要填充时所用的一个字符：

```
class ios_base {
public:
    // ...
    streamsize width() const;           // 取得域宽
    streamsize width(streamsize wide); // 设置域宽
    // ...
};

template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    Ch fill() const;                   // 取得填充字符
    Ch fill(Ch ch);                    // 设置填充字符
    // ...
};
```

width() 函数描述下一次标准库 << 输出操作的最小字符个数，包括数值、*bool*、C风格字符串、字符、指针（21.2.1节）、*string*（20.3.15节）和*bitset*（17.5.3.3节）。例如，

```
cout.width(4);
cout << 12;
```

将打印出前面放了两个空格的12。

“填充”字符用**fill()**函数描述，例如：

```
cout.width(4);
cout.fill('#');
cout << "ab";
```

将给出输出 **##ab**。

默认的填充字符是空格，默认的域宽为0，表示“如所需的那么多字符”。可以用

```
cout.width(0); // “如所需的那么多字符”
```

的方式将域宽设置回默认值，调用函数**width(n)**将最小输出字符数设置为**n**。如果提供的字符更多，它们都会被打印出来。例如，

```
cout.width(4);
cout << "abcdef";
```

产生出**abcdef**而不是**abcd**。一般而言，让正确输出看起来不那么漂亮也比让错误输出看起来十分完美更好一些（另见21.10[21]）。

width(n)只影响紧随其后的一次 << 输出操作：

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

这将产生 **##12:13**，而不是 **##12###:##13**。假设**width(4)**能作用于一系列输出，就会出现后一种情况。如果想让一系列输出都受到**width()**的影响，那我们就必须用**width()**去为每个值刻画宽度。

标准操控符（21.4.6.2节）为描述输出域的宽度提供了一种更优美的方式。

21.4.5 域的调整

在一个域内的字符调整通过调用`setf()`进行控制:

```
cout.setf(ios_base::left, ios_base::adjustfield);    // 居左
cout.setf(ios_base::right, ios_base::adjustfield);   // 居右
cout.setf(ios_base::internal, ios_base::adjustfield); // 中间
```

这些函数在由`ios_base::width()`定义的输出域内设置输出的调整方式,且不影响流状态的任何其他部分。

调整可以以

```
cout.fill('#');

cout << '(';
cout.width(4);
cout << -12 << " ", (");

cout.width(4);
cout.setf(ios_base::left, ios_base::adjustfield);
cout << -12 << " ", (");

cout.width(4);
cout.setf(ios_base::internal, ios_base::adjustfield);
cout << -12 << " )";
```

的方式描述,这将产生`(#-12)`、`(-12#)`、`(-#12)`。中间调整将填充字符放在符号和值之间。如上所示,居右调整是默认方式。如果同时设置了多于一个调整标志会出现什么情况是没有定义的。

21.4.6 操控符

为使程序员能够不必直接以标志位的方式去处理流的状态,标准库提供了另外一组函数,专门用于操控这些状态。这里的关键思想就是在读写对象之间插入一个修改状态的操作。举例来说,我们可以显式地要求刷新输出缓冲区:

```
cout << x << flush << y << flush;
```

在这里, `cout.flush()` 将被调用适当的次数。完成此事是通过 `<<` 的一个版本,它以函数指针为参数,并伺机调用它:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    // ...

    basic_ostream& operator<< (basic_ostream& (*f) (basic_ostream&)) { return f(*this); }
    basic_ostream& operator<< (ios_base& (*f) (ios_base&));
    basic_ostream& operator<< (basic_ios<Ch, Tr>& (*f) (basic_ios<Ch, Tr>&));

    // ...
};
```

为使这种机制能如愿以偿,所用的函数必须是类型正确的非成员函数或静态成员函数。特别地, `flush()` 的定义如下:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch, Tr>& flush(basic_ostream<Ch, Tr>& s)
{
    return s.flush();    // 调用ostream的flush()
}
```

这个声明保证

```
cout << flush;
```

将被解析为

```
cout.operator<< (flush);
```

它调用

```
flush(cout);
```

它进而又调用

```
cout.flush();
```

(在编译时)完成了整个的这一团废话就是为了让我们在使用`cout << flush`的形式时,实际能去调用`basic_ostream::flush()`。

存在着很广泛的一类操作,都是我们可能希望在某个输入或者输出操作之前或者之后执行的。例如:

```
cout << x;
cout.flush();
cout << y;
```

```
cin.unsetf(ios_base::skipws)    // 不要跳过空白
cin >> x;
```

在将这些操作写成独立的语句时,操作间的逻辑联系是不明显的。而失去了逻辑联系将使代码变得更难阅读。提出操控符的概念也就是希望能将`flush()`和`unsetf(ios_base::skipws)`这类操作直接插入输入或者输出操作列表里。例如:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

请注意,操控符都在`std`名字空间里,所以,在`std`不是当前作用域的一部分的地方,就需要对这些名字进行显式限定:

```
std::cout << endl;    // 错误: endl不在作用域
std::cout << std::endl; // ok
```

自然, `basic_istream`也以与`basic_ostream`类似的方式为调用操控符提供了 `>>` 运算符:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    basic_istream& operator>> (basic_istream& (*pf) (basic_istream&));
    basic_istream& operator>> (basic_ios<Ch, Tr>& (*pf) (basic_ios<Ch, Tr>&));
    basic_istream& operator>> (ios_base& (*pf) (ios_base&));
    // ...
};
```

21.4.6.1 带参数的操控符

带参数的操控符也很有用。例如,我们可能想写:

```
cout << setprecision(4) << angle;
```

要求用4位精度打印浮点数变量`angle`的值。

为完成此事, `setprecision`必须返回一个用4初始化过的对象,在它被调用时就能去调用

`cout.precision(4)`。这样的操控符是一个函数对象，它们被 `<<` 调用（而不是通过 `()` 调用）。这个函数对象的具体类型由实现确定，但它可能像下面这样定义：

```
struct smanip {
    ios_base& (*f) (ios_base&, int);    // 要调用的函数
    int i;

    smanip(ios_base& (*ff) (ios_base&, int), int ii) : f(ff), i(ii) { }
};

template<class Ch, class Tr>
ostream<Ch, Tr>& operator<< (ostream<Ch, Tr>& os, const smanip& m)
{
    return m.f(os, m.i);
}
```

`smanip` 的构造函数将其参数存在 `f` 和 `i` 中，`operator<<` 调用 `f(i)`。现在我们可以用以下方式定义 `setprecision()` 了：

```
ios_base& set_precision(ios_base& s, int n)    // 协助函数
{
    return s.precision(n);    // 调用成员函数
}

inline smanip setprecision(int n)
{
    return smanip(set_precision, n);    // 创建函数对象
}
```

我们现在就可以写：

```
cout << setprecision(4) << angle ;
```

程序员也可以根据需要，按照 `smanip` 的风格定义新的操控符（21.10[22]），这样做时并不需要修改标准库模板或者类的定义，如 `basic_istream`、`basic_ostream`、`basic_ios` 和 `ios_base`。

21.4.6.2 标准 I/O 操控符

标准库针对各种各样的格式状态和状态变化提供了一批操控符。标准操控符定义在名字空间 `std` 里。针对 `ios_base` 的操控符由 `<ios>` 给出。针对 `istream` 和 `ostream` 的操控符分别在 `<istream>` 和 `<ostream>` 里给出，同时都在 `<iostream>` 里给出，其他标准操控符则由 `<iomanip>` 给出。

```
ios_base& boolalpha(ios_base&);    // 用符号形式表示真和假（输入和输出）
ios_base& noboolalpha(ios_base& s); // s.unsetf(ios_base::boolalpha)

ios_base& showbase(ios_base&);    // 输出八进制加前缀0，十六进制加前缀0x
ios_base& noshowbase(ios_base& s); // s.unsetf(ios_base::showbase)

ios_base& showpoint(ios_base&);
ios_base& noshowpoint(ios_base& s); // s.unsetf(ios_base::showpoint)

ios_base& showpos(ios_base&);
ios_base& noshowpos(ios_base& s); // s.unsetf(ios_base::showpos)

ios_base& skipws(ios_base&);    // 跳过空白
ios_base& noskipws(ios_base& s); // s.unsetf(ios_base::skipws)

ios_base& uppercase(ios_base&);    // 用X和E而不是x和e
ios_base& nouppercase(ios_base&); // 用x和e而不是X和E
```

```

ios_base& internal (ios_base&);           // 调整 (21.4.5节)
ios_base& left (ios_base&);               // 值后填充
ios_base& right (ios_base&);              // 值前填充

ios_base& dec (ios_base&);                // 整数基数10 (21.4.2节)
ios_base& hex (ios_base&);                // 整数基数16
ios_base& oct (ios_base&);                // 整数基数8

ios_base& fixed (ios_base&);              // 浮点格式dddd.dd (21.4.3节)
ios_base& scientific (ios_base&);         // 科学格式d.ddddEdd

template <class Ch, class Tr>
    basic_ostream<Ch, Tr>& endl (basic_ostream<Ch, Tr>&); // 放 '\n' 并刷新
template <class Ch, class Tr>
    basic_ostream<Ch, Tr>& ends (basic_ostream<Ch, Tr>&); // 放 '\0'
template <class Ch, class Tr>
    basic_ostream<Ch, Tr>& flush (basic_ostream<Ch, Tr>&); // 刷新流

template <class Ch, class Tr>
    basic_istream<Ch, Tr>& ws (basic_istream<Ch, Tr>&);    // 吃掉空白

smanip resetiosflags (ios_base::fmtflags f); // 清标志 (21.4节)
smanip setiosflags (ios_base::fmtflags f);   // 设置标志 (21.4节)
smanip setbase (int b);                      // 基于b输出整数 (21.4.2节)
smanip setfill (int c);                      // 用c作为填充字符 (21.4.4节)
smanip setprecision (int n);                 // n位数字 (21.4.3节、21.4.6节)
smanip setw (int n);                         // 下个域宽为n个字符 (21.4.4节)

```

例如,

```
cout << 1234 << ' ' << hex << 1234 << ' ' << oct << 1234 << endl;
```

将产生1234, 4d2, 2322, 且

```
cout << ' (' << setw(4) << setfill('#') << 12 << " ) (" << 12 << " )\n";
```

将产生(##12)(12)。

如果用的是不带参数的操控符, 那么就不要加括号。要使用带参数的标准操控符, 应记住写 `#include <iomanip>`。例如,

```

#include <iostream>
using namespace std;

int main()
{
    cout << setprecision(4)           // 错误: setprecision未定义 (忘记包含 <iomanip>)
        << scientific()              // 错误: ostream<<ostream& (误用括号)
        << 1.41421 << endl;
}

```

21.4.6.3 用户定义操控符

程序员也可以按照标准的风格添加自己的操控符。这里我要展示另外一种风格, 我发现它对于浮点数的格式化非常有用。

`precision`对所有输出操作具有持续性的作用, 而`width()`只对下一个数值输出操作起作用。我希望的是某种东西, 它能使按照某种预先定义的格式输出一个浮点数的事情变得非常简单, 但又不影响在这个流上随后的输出操作。这里的基本想法就是定义一个表示格式的类, 用另一个类表示一个格式再加一个需要格式化的值, 而后让运算符 `<<` 按照这个格式向一个 `ostream` 输出这个值。例如,

```

Form gen4(4); // 一般格式, 精度为4
void f(double d)
{
    Form sci8 = gen4;
    sci8.scientific().precision(8); // 科学格式, 精度为8
    cout << d << ' ' << gen4(d) << ' ' << sci8(d) << ' ' << d << '\n';
}

```

调用f(1234.56789)将写出

```
1234.57 1235 1.23456789e+03 1234.57
```

注意, **Form**在这里的使用并没有影响流的状态, 所以, *d*的最后一个输出还是具有某种默认格式, 与第一个完全一样。

下面是一个简化的实现:

```

class Bound_form; // Form加值
class Form {
    friend ostream& operator<<(ostream&, const Bound_form&);

    int prc; // 精度
    int wdt; // 宽度, 0表示按需要的宽度
    int fmt; // 一般、科学或定点格式(21.4.3节)
    // ...

public:
    explicit Form(int p = 6) : prc(p) // 默认精度为6
    {
        fmt = 0; // 一般格式(21.4.3节)
        wdt = 0; // 按需要的宽度
    }

    Bound_form operator()(double d) const; // 对*this和d创建一个Bound_form

    Form& scientific() { fmt = ios_base::scientific; return *this; }
    Form& fixed() { fmt = ios_base::fixed; return *this; }
    Form& general() { fmt = 0; return *this; }

    Form& uppercase();
    Form& lowercase();
    Form& precision(int p) { prc = p; return *this; }

    Form& width(int w) { wdt = w; return *this; } // 应用于所有类型
    Form& fill(char);

    Form& plus(bool b = true); // 显式加
    Form& trailing_zeros(bool b = true); // 打印尾随的0
    // ...
};

```

这里的想法是, 让**Form**保存格式化一个数据项所需的所有信息。默认情况的选择对于大部分用户是合理的, 各种成员函数可用于重新设置格式的各个方面。运算符()建立起一个值和用于输出它的格式的关联。此后, 就可以通过适当的<<函数向指定的流输出这样建立起的**Bound_form**:

```

struct Bound_form {
    const Form& f;
    double val;
};

```

```

    Bound_form(const Form& ff, double v) : f(ff), val(v) {}
};

Bound_form Form::operator()(double d) const { return Bound_form(*this, d); }

ostream& operator<<(ostream& os, const Bound_form& bf)
{
    ostringstream s;           // 字符串流在21.5.3节介绍
    s.precision(bf.f.prc);
    s.setf(bf.f.fmt, ios_base::floatfield);
    s << bf.val;                // 在s里组合输出字符串
    return os << s.str();       // 将s输出到os
}

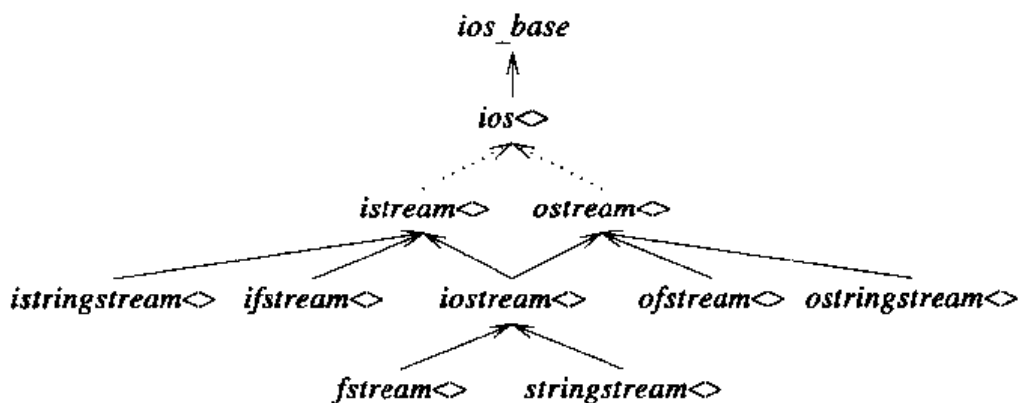
```

写出一个不这么简化的 << 实现的工作留做练习 (21.10[21])。Form和Bound_form类很容易扩展，以便用于整数、字符串等的格式化工作 (21.10[20])。

请注意，这些声明实际上将 << 和 () 的组合看成一个三元函数，cout << sci4(d) 将一个 ostream、一个格式和一个值汇集到一起，而后才去做实际的计算。

21.5 文件流与字符串流

当一个C++ 程序开始工作时，cout、cerr、clog和cin以及它们对应的宽字符流 (21.2.1 节) 都已经可以用了。这些流都是默认设置好的，而且它们所对应的I/O设备或文件由“系统”确定。除此之外，你还可以创建自己的流，这时你就必须说明将流附着到哪里。将流附着于文件或者string都是很常见的情况，因此标准库也直接支持这些操作。下面是标准流类的层次结构图：



写了 <> 后缀的类是以字符类型为参数的模板，它们的名字都以basic_ 作为前缀。点箭头表示虚基类 (15.2.4节)。

文件和串都是你可以从其中读、向里面写的容器实例。因此，你也可以有一个同时支持 << 和 >> 操作的流。这种流被称为iostream，它也定义在名字空间std里，并在头文件 <iostream> 里描述：

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_iostream : public basic_istream<Ch, Tr>, public basic_ostream<Ch, Tr> {
public:
    explicit basic_iostream(basic_streambuf<Ch, Tr>* sb);
    virtual ~basic_iostream();
};

```

```
typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

对一个*iostream*的读写通过*iostream*中*streambuf*的写缓冲区和读缓冲区操作进行控制（21.6.4节）。

21.5.1 文件流

下面是一个完整的程序，它将一个文件复制到另一个文件。文件名从命令行参数获得：

```
#include <fstream>
#include <cstdlib>
void error(const char* p, const char* p2 = "")
{
    cerr << p << ' ' << p2 << '\n';
    std::exit(1);
}

int main(int argc, char* argv[])
{
    if (argc != 3) error("wrong number of arguments");

    std::ifstream from(argv[1]);           // 打开输入文件流
    if (!from) error("cannot open input file", argv[1]);

    std::ofstream to(argv[2]);             // 打开输出文件流
    if (!to) error("cannot open output file", argv[2]);

    char ch;
    while (from.get(ch)) to.put(ch);       // 复制字符

    if (!from.eof() || !to) error("something strange happened");
}
```

为输入而打开一个文件，也就是以这个文件的名字为参数创建一个类*ifstream*（输入文件流）的对象。与此类似，为输出而打开一个文件就是以文件名为参数创建一个类*ofstream*（输出文件流）的对象。在这两种情况下，我们都要检测所创建对象的状态，查看有关文件是否成功打开。

*basic_ofstream*的声明在 *<fstream>* 里大致如下：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ofstream : public basic_ostream<Ch, Tr> {
public:
    basic_ofstream();
    explicit basic_ofstream(const char* p, openmode m = out);

    basic_filebuf<Ch, Tr>* rdbuf() const;    // 取得指向当前文件缓冲区的指针（21.6.4节）

    bool is_open() const;
    void open(const char* p, openmode m = out);
    void close();
};
```

*basic_ifstream*与*basic_ofstream*类似，但它是从*basic_istream*派生的，按照默认约定是为了读入而打开的。此外，标准库还提供了*basic_fstream*，它也很像*basic_ofstream*，但却是从*basic_iostream*派生的，按照默认方式是既可以读也可以写。

如常，已用*typedef*定义最常用的类型：

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
```

```
typedef basic_fstream<char> fstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_fstream<wchar_t> wfstream;
```

文件流构造函数有第二个参数，用于描述打开的不同模式：

```
class ios_base {
public:
    // ...

    typedef implementation_defined3 openmode;
    static openmode app,      // 附加
        ate,                 // 打开并找到文件尾（读做“at end”）
        binary,              // 采用二进制模式I/O（不是文本模式）
        in,                  // 为读而打开
        out,                 // 为写而打开
        trunc;               // 将文件截断到0长度

    // ...
};
```

*openmode*的实际值及其实际意义由具体实现确定。有关细节请查看你的系统和库的手册，并做些试验。上面注释应当给了你有关各种模式意义的一些线索。例如，我们可以打开一个文件，并使向它写入的内容都附在文件的最后

```
ofstream mystream(name.c_str(), ios_base::app);
```

也可以打开一个既能输入也能输出的文件，例如，

```
fstream dictionary("concordance", ios_base::in | ios_base::out);
```

21.5.2 流的关闭

通过对一个文件的流调用*close()*，就可以显式地关闭这个文件：

```
void f(ostream& mystream)
{
    // ...

    mystream.close();
}
```

然而，流的析构函数也会隐式地完成这件事情。所以，只是在某个文件必须在达到它的流变量声明的作用域结束前关闭时，才需要显式地调用*close()*。

由此产生了一个问题，实现怎么能保证预定义的流*cout*、*cin*、*cerr*和*clog*能够在它们被第一次使用之前创建，并在它们的最后使用之后（才）关闭呢？自然，*<iostream>*流库的不同实现可能采用不同的技术来达到这种效果。这件事如何完成毕竟是实现中的一项细节，对用户应该是不可见的。在这里，我要给出一种足够通用的技术，它能保证各种类型的全局变量都能按照正确的顺序构造与析构。一个具体实现有可能利用编译器或连接器的特殊性质把这件事情做得更好些。

最基本的想法就是定义一个协助类，它是一个计数器，跟踪着 *<iostream>* 曾被多少次包含到一个分别编译源文件里：

```
class ios_base::Init {
    static int count;
```

```

public:
    Init();
    ~Init();
};

namespace { ios_base::Init __ioint; } // 放在 <iostream>, 每个 #include <iostream> 的文件都有一个副本
int ios_base::Init::count = 0;        // 在某个 .c 文件里

```

在每个编译单位（9.1节）里声明其自身的一个名字为 `__ioint` 的对象。对象 `__ioint` 的构造函数利用 `ios_base::Init::count` 作为首次开关，以确保流I/O库的全局变量恰好做一次初始化：

```
ios_base::Init::Init() { if (count++ == 0) { /* 初始化cout、cerr、cin等 */ } }
```

与此相对应，`__ioint` 对象的析构函数利用 `ios_base::Init::count` 作为末次开关来确保这些流都能关闭：

```
ios_base::Init::~Init() { if (--count == 0) { /* 清理cout（刷新等）、cerr、cin等 */ } }
```

这是一种通用技术，可用于处理包含需要初始化和最后清理的全局对象的库。在一个执行期间所有代码都居于内存中的系统里，这种技术几乎没有额外开销。当情况不是这样时，将每个目标文件装进内存以便执行其中的初始化代码，其额外开销就可能很显著。所以，最好是尽可能地避免全局对象。对于某个类，如果它的各个操作都执行了很多工作，让每个操作都去检测首次开关（像 `ios_base::Init::count` 那样）以确保初始化也是一种很合理的方式。但是，对于流而言，这种方式则因为代价过于高昂而无法使用。要求读写一个字符的函数都去检测首次开关造成的额外开销太大了。

21.5.3 字符串流

也可以将流附着在 `string` 上，也就是说，我们可以通过流所提供的格式化功能从 `string` 读或者向 `string` 里写。这样的流称为 `stringstream`，它们在 `<sstream>` 里定义：

```

template <class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_stringstream : public basic_istream<Ch, Tr> {
public:
    explicit basic_stringstream(ios_base::openmode m = out|in);
    explicit basic_stringstream(const basic_string<Ch, Tr, A>& s, ios_base::openmode m = out|in);

    basic_string<Ch, Tr, A> str() const;           // 获取string的副本
    void str(const basic_string<Ch, Tr, A>& s);     // 将值设置为s的副本

    basic_stringbuf<Ch, Tr, A>* rdbuf() const;    // 取得到当前文件缓冲区的指针
};

```

一个 `basic_istringstream` 就像是一个 `basic_stringstream`，但它是由 `basic_istream` 派生，默认是为读而打开的；一个 `basic_ostringstream` 也像是 `basic_stringstream`，但它是由 `basic_ostream` 派生，默认是为写而打开的。

通常，通过 `typedef` 提供了最常用的一组专门化：

```

typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;

```

例如, *ostringstream* 可以用于对消息串做格式化:

```
string compose(int n, const string& cs)
{
    extern const char* std_message[];
    ostringstream ost;
    ost << "error(" << n << ") " << std_message[n] << " (user comment: " << cs << ')';
    return ost.str();
}
```

在这里不需要检查溢出, 因为 *ost* 将根据需要自动扩展。如果需要处理一些格式化要求, 其中的情况比通常基于文本行的输出设备能做的事情更复杂时, 这一技术就非常有用。

对于字符串流初始值的处理与文件流对其文件的处理类似:

```
string compose2(int n, const string& cs) // 等价于compose()
{
    extern const char* std_message[];
    ostringstream ost("error(", ios_base::ate); // 从初始化串的最后开始写
    ost << n << ") " << std_message[n] << " (user comment: " << cs << ')';
    return ost.str();
}
```

一个 *istringstream* 是一个输入流, 用于从其初始化的 *string* 值中读入 (就像一个 *ifstream* 由其文件读入一样):

```
#include <sstream>

void word_per_line(const string& s) // 每行输出一个词
{
    istringstream ist(s);
    string w;
    while (ist >> w) cout << w << '\n';
}

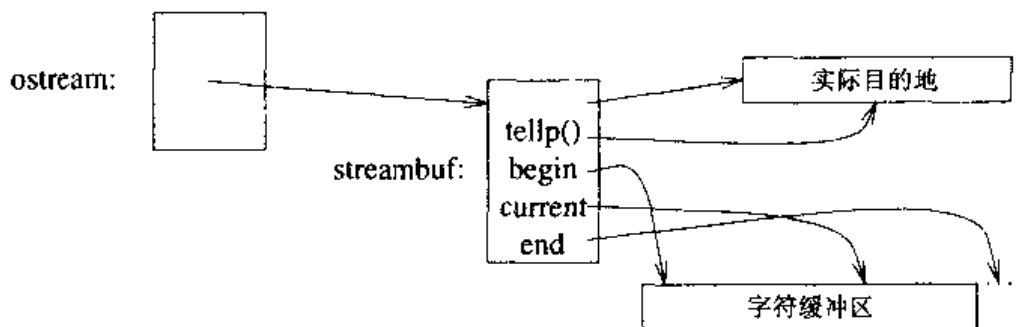
int main()
{
    word_per_line("If you think C++ is difficult, try English");
}
```

初始化表达式 *string* 被复制进 *istringstream*, 串的开始也将使输入终止。

也可以定义流, 使它直接从字符数组读或者直接向字符数组里写 (21.10[26])。在处理老代码时经常需要这样做, 特别是因为 *ostrstream* 和 *istrstream* 做的就是这些, 它们是原始流库的组成部分。

21.6 缓冲

从概念上看, 一个输出流总是将某些字符放入一个缓冲区。在此后的某个时刻, 这些字符才被写到它们真正应该去的地方。这样的缓冲区称为一个 *streambuf* (21.6.4节), 它的定义可以在 *<streambuf>* 里找到。不同的 *streambuf* 实现不同的缓冲策略。在典型情况下, *streambuf* 将字符保存在一个数组里, 直到出现上溢, 才迫使它将这些字符写到它们的实际目的地去。这样, 一个 *ostream* 可以用下面的图形表示:



*ostream*和它的*streambuf*必须有着同样的一组模板参数，并由此确定字符缓冲区所用的字符类型。

*istream*的情况也类似，除了字符的流向相反之外。

非缓冲I/O也就是一种I/O，其中的流缓冲立即传送每一个字符，而不是保存着这些字符，直到积累了足够高效传输的一组字符后再行传送。

21.6.1 输出流和缓冲区

*ostream*提供了一些操作，它们能依据规则（21.2.1节）或者显式的转换指令，将各种类型的值转换为字符序列。此外，*ostream*还提供了一些直接处理它的*streambuf*的操作：

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    explicit basic_ostream(basic_streambuf<Ch, Tr>* b);

    pos_type tellp(); // 取当前位置
    basic_ostream& seekp(pos_type); // 设置当前位置
    basic_ostream& seekp(off_type, ios_base::seekdir); // 设置当前位置

    basic_ostream& flush(); // 腾空缓冲区（送到实际目的地）

    basic_ostream& operator<<(basic_streambuf<Ch, Tr>* b); // 从b写
};
  
```

*ostream*被构造时需要有一个*streambuf*参数，它确定了对写入字符的处理方式及其最终流向。举例来说，*ostream*（21.5.3节）或*ofstream*（21.5.1节）的创建都需要用一个合适的*streambuf*（21.6.4节）去初始化*ostream*。

*seekp()*函数用于在*ostream*里为写入做定位，*p*后缀表示这是为向流中放入（putting）字符而做的定位。除非有关的流确实附着在某种使定位操作有意义的东西上（例如是一个文件），否则这些函数将没有影响。*pos_type*表示文件里的一个字符位置，*off_type*则表示从某个由*ios_base::seekdir*标明的点出发的偏移量：

```

class ios_base {
    // ...
    typedef implementation_defined seekdir;
    static const seekdir beg, // 从当前文件起始处出发去找
        cur, // 从当前位置出发去找
        end; // 从当前文件最后反向找
    // ...
};
  
```

流定位从0开始，因此我们可以把一个文件想象为一个包含*n*个字符的数组。例如，

```
int f(ofstream& fout) // fout引用某个文件
{
    fout.seekp(10);
    fout << '#';           // 加入字符并改变位置(+1)
    fout.seekp(-2, ios_base::cur);
    fout << '*';
}
```

这将会把一个 # 放进 `file[10]`，将一个 * 放进 `file[9]`。对于普通的 `istream` 或 `ostream` 的元素，就没有与此类似的随机访问方式（参见 21.10[13]）。企图查到文件的开头或者结束位置之外，都将把流推入 `bad()` 状态（21.3.3 节）。

`flush()` 操作使用户可以随时腾空缓冲区，不需要等待溢出。

可以用 `<<` 直接把一个 `streambuf` 写进一个 `ostream`，这主要是作为提供给 I/O 机制实现者的一种方便手段。

21.6.2 输入流和缓冲区

`istream` 提供了一些操作，用以读入字符并将它们转换为各种类型的值（21.3.1 节）。此外，`istream` 也提供了一些直接处理其 `streambuf` 的操作：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr> {
public:
    // ...
    explicit basic_istream(basic_streambuf<Ch, Tr>* b);

    pos_type tellg();           // 取当前位置
    basic_istream& seekg(pos_type); // 设置当前位置
    basic_istream& seekg(off_type, ios_base::seekdir); // 设置当前位置

    basic_istream& putback(Ch c); // 将c放回缓冲区
    basic_istream& unget();       // 将最近读的字符放回
    int_type peek();             // 查看下一个要读的字符

    int sync();                  // 清除缓冲区（刷新）

    basic_istream& operator>>(basic_streambuf<Ch, Tr>* b); // 读到b
    basic_istream& get(basic_streambuf<Ch, Tr>& b, Ch t = Tr::newline());
    streamsize readsome(Ch* p, streamsize n); // 读至多n个字符
};
```

定位函数的功能与 `ostream` 里相应的函数类似（21.6.1 节）。`g` 后缀表示这是为从流中取出（getting）字符而做的定位。这些 `p` 和 `g` 后缀是需要的，因为我们可以创建从 `istream` 和 `ostream` 一起派生的一个 `iostream`，对于这种流就需要保持一个放入的定位和一个取出的定位。

`putback()` 函数使程序可以将一个目前不需要的字符放回流里，以便后面某个时刻再去读，如 21.3.5 节所示。`unget()` 函数将最后读入的字符退回。不幸的是，在输入流上向后退并不总是可行的，例如，想退到读入第一个字符之前的再前面一个字符，就会引起设置 `ios_base::failbit`。你在一次成功的读入之后可以退回一个字符，这样做是有保证的。`peek()` 读下一个字符，但将它留在 `streambuf` 里以便将来再去读。这样，`c = peek()` 也就等价于 `(c = get(), unget(), c)` 和 `(putback(c = get()), c)`。注意，设置 `ios_base::failbit` 有可能触发一个异常（21.3.6 节）。

刷新 `istream` 的操作通过 `sync()` 完成，但是未必总能做得正确。对于某些种类的流，我们

可能不得不从实际信息源中重新读入字符。这未必总能做到，也不一定是真正希望做的。因此，在`sync()`成功时将返回0，如果失败就设置`ios_base::badbit`并返回-1。这样，遇到设置了`badbit`的时候就可能触发一个异常（21.3.6节）。对于附着在`ostream`上的流的`sync()`将为输出刷新缓冲区。

以`streambuf`为目标的`>>`和`get()`操作主要供I/O机制的实现者使用，只有这些实现者应该去直接操作`streambuf`。

`readsome()`函数是一种低级操作，它使用户可以向流里窥视，看看那里是否还有能读的字符。当你不希望等待输入（例如来自键盘的输入）时，这种功能就很有用。参见`in_avail()`（21.6.4节）。

21.6.3 流和缓冲区

流与其缓冲区之间的联系是由流的`basic_ios`维护的：

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
public:
    // ...
    basic_streambuf<Ch, Tr>* rdbuf() const;           // 取得缓冲区
    // set buffer, clear(), and return pointer to old buffer:
    basic_streambuf<Ch, Tr>* rdbuf(basic_streambuf<Ch, Tr>* b);

    locale imbue(const locale& loc);                 // 设置现场 (locale, 并返回原来的现场)

    char narrow(char_type c, char d) const;          // 从char_type的c做出char值
    char_type widen(char c) const;                   // 从char类型的c做出char_type值
    // ...
protected:
    basic_ios();
    void init(basic_streambuf<Ch, Tr>* b);           // 设置初始缓冲区
};
```

除了读和设置流的`streambuf`（21.6.4节）之外，`basic_ios`还提供了`imbue()`，用于读出一个流的现场和重新设置其现场（21.7节），方式是对其基类`ios_base`调用`imbue()`（21.7.1节），以及对流的缓冲区调用`pubimbue()`（21.6.4节）。

`narrow()`和`widen()`函数用于在`char`类型和缓冲区的`char_type`类型之间的转换。`narrow(c, d)`的第二个参数是一个`char`，当不存在对应于`char_type`类型值`c`的`char`时，就返回作为第二个参数的`char`。

21.6.4 流缓冲区

I/O操作的描述中并没有提及文件的类型，但在相对于各种缓冲策略方面，也不是所有设备都应一致地对待。例如，关联于`string`的`ostream`（21.5.3节）所需要的缓冲区种类就与关联于文件的`ostream`（21.5.3节）不同。处理这些问题的方式就是在初始化时为不同的流提供不同的缓冲区。对各种缓冲区只存在着的一组操作，因此，`ostream`函数中没有包含能将这种流区分开的代码。不同类型的缓冲区都是由`streambuf`类派生的，类`streambuf`在出现不同缓冲区策略的地方提供了一些虚函数，例如那些处理缓冲区上溢和下溢的函数。

`basic_streambuf`类提供了两个界面，其公用界面的主要目标是流类（如`istream`、`ostream`、

fstream、*stringstream*) 的实现者。此外还有了一个保护界面, 供新的缓冲策略和新的输入源或输出目标的 *streambuf* 的实现者使用。

为了理解 *streambuf*, 首先考虑由保护界面所提供的缓冲区基础模型将很有帮助。假定这个 *streambuf* 有一个放入区, << 向里面写; 还有一个取出区, >> 从里面读。每个区域都用三个指针描述: 一个开始指针、一个当前位置指针和一个末端加一位置指针。这些指针都通过函数进行操作:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:

    Ch* eback() const;           // “取出缓冲区” 的开始
    Ch* gptr() const;           // 已填入的下一字符 (读下一个char取自这里)
    Ch* egptr() const;          // 取出缓冲区的末端加一位置

    void gbump(int n);           // 给gptr() 加n
    void setg(Ch* begin, Ch* next, Ch* end); // 设置eback(), gptr() 和egptr()

    Ch* pbase() const;          // “放入缓冲区” 的开始
    Ch* pptr() const;           // 下一空字符位 (写入的下一个字符放在这里)
    Ch* epptr() const;          // 放入缓冲区的末端加一位置
    void pbump(int n);           // 给pptr() 加n
    void setp(Ch* begin, Ch* end); // 设置pbase() 和pptr() 到开头, epptr() 到末尾
    // ...
};
```

给了一个字符数组, *setg()* 和 *setp()* 可以正确地设置各个指针。某个实现可能按如下方式访问其取出区:

```
template <class Ch, class Tr = char_traits<Ch> >
basic_streambuf<Ch, Tr>::int_type basic_streambuf<Ch, Tr>::snextc()
// 跳过当前字符, 读下一个字符
{
    if (1 < egptr() - gptr()) { // 如果缓冲区里至少有两个字符
        gbump(1);              // 跳过当前字符
        return Tr::to_int_type(*gptr()); // 返回新的当前字符
    }
    if (1 == egptr() - gptr()) { // 如果缓冲区里只有一个字符
        gbump(1);              // 跳过当前字符
        return underflow();
    }
    // 缓冲区空 (或者不存在缓冲区), 试着填充它:
    if (Tr::eq_int_type(uflow(), Tr::eof())) return Tr::eof();
    if (0 < egptr() - gptr()) return Tr::to_int_type(*gptr()); // 返回新的当前字符
    return underflow();
}
```

通过 *gptr()* 访问缓冲区, *egptr()* 标明了取缓冲区的边界。*uflow()* 和 *underflow()* 从实际信息源中读入字符。对 *traits_type::to_int_type()* 的调用保证了这些代码并不依赖于实际的字符类型。这些代码允许各种流缓冲区类型, 并考虑了虚函数 *uflow()* 和 *underflow()* 或许决定引进新的取出区 (通过 *setg()*) 的可能性。

streambuf 的公用界面如下:

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
```

```

public:
    // 普通的typedef (21.2.1节)
    virtual ~basic_streambuf();

    locale pubimbue(const locale &loc);           // 设置locale (并取出原locale)
    locale getloc() const;                       // 取出locale

    basic_streambuf* pubsetbuf(Ch* p, streamsize n); // 设置缓冲区空间

    pos_type pubseekoff(off_type off, ios_base::seekdir way,          // 定位 (21.6.1节)
                        ios_base::openmode m = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type p, ios_base::openmode m = ios_base::in | ios_base::out);

    int pubsync();                               // sync(); 见21.6.2节

    int_type snextc();                            // 跳过当前字符, 而后取下一个字符
    int_type sbumpc();                            // 将gptr()推进1
    int_type sgetc();                             // 取当前字符
    streamsize sgetn(Ch* p, streamsize n);        // 取进p[0]..p[n-1]

    int_type sputbackc(Ch c);                     // 将c放回缓冲区 (21.6.2节)
    int_type sungetc();                           // 退回最近的一个字符

    int_type sputc(Ch c);                         // 放入c
    streamsize sputn(const Ch* p, streamsize n); // 放入p[0]..p[n-1]
    streamsize in_avail();                        // 有输入可用吗?
    // ...
};

```

公共界面包含将字符插入缓冲区和从缓冲区提取字符的函数。这些函数都很简单, 很容易在线化, 这些对效率都极其重要。

位于特定缓冲策略的各个实现部分中的那些函数将调用保护界面中对应的函数。例如, `pubsetbuf()` 调用 `setbuf()`, 这个函数将由某个派生类覆盖, 以实现在这个类里为缓冲它的字符提供相应的存储。采用两个函数实现诸如 `setbuf()` 这样的操作, 使 `iostream` 的实现者可以在用户代码之前和之后完成一些“簿记性”工作。举例来说, 实现者可能用一个 `try` 块将对虚函数的调用包起来, 并捕捉用户代码抛出的异常。

按照默认约定, `setbuf(0, 0)` 表示“不缓冲”; `setbuf(p, n)` 表示用 `p[0]..p[n-1]` 保存缓冲存储的字符。

对 `in_avail()` 的调用可以查看在缓冲区里还存在多少可用的字符, 这可以用于避免等待输入。当我们从一个连接到键盘的流中读入时, `cin.get(c)` 可能要一直等到用户从吃饭的地方回来。在某些系统上, 对于某些应用, 在读入时应该将这种情况考虑在内。例如:

```

if (cin.rdbuf()->in_avail()) { // get() 将不会阻塞
    cin.get(c);
    // 做某些事
}
else {                          // get() 可能阻塞
    // 做另一些事
}

```

请注意, 在某些系统里, 确定是否存在输入也可能很困难。这样就可能出现在某些情况下 `in_avail()` 的 (不良) 实现返回了 0, 而这时实际输入操作原来可能成功。

除了供 `basic_istream` 和 `basic_ostream` 使用的公用界面外, `basic_streambuf` 还提供了一个保护界面, 供 `streambuf` 的实现者使用。那些确定策略的虚函数的声明就放在这里:

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_streambuf {
protected:
    // ...

    basic_streambuf();

    virtual void imbue(const locale &loc);           // 设置locale

    virtual basic_streambuf* setbuf(Ch* p, streamsize n);

    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode m = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type p,
                             ios_base::openmode m = ios_base::in | ios_base::out);

    virtual int sync();                             // sync(); 见21.6.2节
    virtual int showmanyc();
    virtual streamsize xsgetn(Ch* p, streamsize n);   // 取n个字符
    virtual int_type underflow(); // 再填充取出区; 返回字符或eof
    virtual int_type uflow();    // 再填充取出区; 返回字符或eof, gptr() 加1

    virtual int_type pbackfail(int_type c = Tr::eof()); // 返回失败

    virtual streamsize xspn(const Ch* p, streamsize n); // 放入n个字符
    virtual int_type overflow(int_type c = Tr::eof());  // 放入区满
};

```

在缓冲区空时调用`underflow()`和`uflow()`函数,它们会从实际输入源取得下一个字符。如果输入源已不再有下一个可用字符了,它们就将这个流设置为`eof`状态(21.3.3节)。如果当时的状态设置不抛出异常,那么就返回`traits_type::eof()`。`uflow()`将`gptr()`加一,使它越过返回的字符,而`underflow()`不这样做。请记住,你的系统里的缓冲区通常比`iostream`库所引入的缓冲区更多,所以,即使你使用非缓冲式流I/O,也可能受到缓冲延迟。^①

在缓冲区满时调用`overflow()`函数,就将字符传送到实际的输出目的地。调用`overflow(c)`将在输出缓冲区内容之后再加上字符`c`。如果无法继续向那个目的地写输出,它就将这个流设置到`eof`状态(21.3.3节)。如果这样做时没有导致抛出异常,那么就返回`traits_type::eof()`。

`showmanyc()` (“显示有多少字符”)函数是一个老函数,意在使用户可以了解机器输入系统状态的某些情况。它返回的是对于还有多少字符能“立即”读入的一个估计值,比如说,这个“立即”可能表示用完操作系统的缓冲区而无需等待磁盘读入。如果`showmanyc()`无法保证在遇到文件结束之前还可能读到任何字符,它的调用就会返回-1。这(必然)是相当低级且与实现高度相关的。在没有仔细阅读过你的系统手册和做过一些试验之前,请不要使用`showmanyc()`。

按照默认约定,每个流都取用全局的现场(21.7节)。调用`pubimbue(loc)`或`imbue(loc)`将使一个流采用`loc`作为它的现场。

为特定种类的流所用的`streambuf`都是从`basic_streambuf`派生的。它需要提供构造函数和初始化函数,以建立`streambuf`与字符的实际来源(实际去处)之间的联系,并覆盖那些确定缓冲策略的虚函数。例如,

① 这里作者希望提醒读者,所用系统在如何处理I/O动作时有它自己的规矩,我们写的C++程序未必能控制它。虽然我们可以要求采用非缓冲式I/O进行输入或输出,但实际I/O也可能是经由系统的某种缓冲机制完成的。
——译者注

```

template <class Ch, class Tr = char_traits<Ch> >
class basic_filebuf : public basic_streambuf<Ch, Tr> {
public:
    basic_filebuf();
    virtual ~basic_filebuf();

    bool is_open() const;
    basic_filebuf* open(const char* p, ios_base::openmode mode);
    basic_filebuf* close();

protected:
    virtual int showmanyc();
    virtual int_type underflow();
    virtual int_type uflow();
    virtual int_type pbackfail(int_type c = Tr::eof());
    virtual int_type overflow(int_type c = Tr::eof());

    virtual basic_streambuf<Ch, Tr>* setbuf(Ch* p, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode m = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type p,
        ios_base::openmode m = ios_base::in | ios_base::out);
    virtual int sync();
    virtual void imbue(const locale& loc);
};

```

操纵缓冲区等的函数直接由***basic_streambuf***继承而来。只有那些影响初始化和缓冲策略的函数需要另行提供。

通常，在这里也提供了一些明显的***typedef***及其宽字符流的对应物：

```

typedef basic_streambuf<char> streambuf;
typedef basic_stringbuf<char> stringbuf;
typedef basic_filebuf<char> filebuf;

typedef basic_streambuf<wchar_t> wstreambuf;
typedef basic_stringbuf<wchar_t> wstringbuf;
typedef basic_filebuf<wchar_t> wfilebuf;

```

21.7 现场

一个***locale***（现场）也就是一个控制不同类别字符的划分（如字母、数字等）、字符串的核对顺序、数值在输入输出中的出现形式等的对象。最常见的情况是，***locale***由***iostream***隐式使用，以保证能够遵循某种自然语言或者文化的习惯和常规。在这种情况下，程序员绝不会出现看到***locale***对象。然而，通过改变***stream***的***locale***对象，程序员就可以改变流的行为方式以设法适应另一组不同的常规。

一个现场也就是类***locale***的一个对象，该类的定义在名字空间***std***里，由 ***<locale>*** 给出（D.2节）。

```

class locale {           // 完整声明在D.2节
public:
    // ...

    locale() throw();           // 复制当前全局locale
    explicit locale(const char* name); // 以C现场name创建locale
    basic_string<char> name() const; // 给出本locale的名字

```

```

    locale(const locale&) throw();           // 复制locale
    const locale& operator=(const locale&) throw(); // 复制locale

    static locale global(const locale&);      // 设置全局locale (并取出原locale)
    static const locale& classic();           // 取出C定义的locale
};

```

使用现场最简单的方式就是从一个现存的现场转到另一个现场。例如，

```

void f()
{
    std::locale loc("POSIX");                // 针对POSIX的标准现场
    cin.imbue(loc);                          // 让cin使用loc
    // ...
    cin.imbue(std::locale());               // 使cin恢复使用默认(全局) locale
}

```

imbue() 函数是**basic_ios**的成员函数(21.7.1节)。

如上所述，某些相当标准的现场以字符串为名字，它们通常是与C语言共享的。也可以设置**locale**，使随后新创建的流都使用它：

```

void g(const locale& loc = locale())        // 默认使用当前全局现场
{
    locale old_global = locale::global(loc); // 使loc成为默认现场
    // ...
}

```

设置全局**locale**不会改变现存的流的行为，它们用的还是全局**locale**原来的值。特别是**cin**、**cout**等都不会受到影响。如果想改变它们，就必须对它们显式地使用**imbue()**。

用一个**locale**去浸染(**imbue**)一个流，将改变其行为的某些刻面(**facet**)。也可以直接用**locale**的某些成员去定义新的**locale**，或者用新的刻面去扩展**locale**。举个例子，可以明确地使用某个**locale**去控制输入和输出中货币单位、日期等的表现形式(21.10[25])，控制编码集之间转换。有关现场的概念(**locale**和**facet**类)以及标准现场和刻面等都将在附录D介绍。

C风格的现场在 `<locale>` 和 `<locale.h>` 里给出。

21.7.1 流回调

有时，人们会希望向流状态中添加一些东西。举例来说，有人可能希望某个流“知道”应该用极坐标还是笛卡儿坐标形式输出**complex**。类**ios_base**提供了一个函数**xalloc()**，专为这种简单状态信息分配空间。由**xalloc()**返回的值标识出一对位置，借助**iword()**和**pword()**可以访问这些位置：

```

class ios_base {
public:
    // ...

    ~ios_base();

    locale imbue(const locale& loc);          // 设置locale并返回原locale; 见D.2.3节
    locale getloc() const;                   // 取得locale

    static int xalloc();                     // 取得一个整数和一个指针 (都初始化为0)
    long& iword(int i);                      // 访问整数iword(i)
    void*& pword(int i);                    // 访问指针pword(i)
}

```



```

// 回调
enum event { erase_event, imbue_event, copyfmt_event }; // 事件类型

typedef void (*event_callback)(event, ios_base&, int i);
void register_callback(event_callback f, int i); // 将f附在iword(i)
};

```

有时，一个实现者或者一个用户可能需要在流状态改变时得到通知，`register_callback()` 函数将“注册”一个在它的“事件”发生时就会调用的函数。这样，当出现调用 `imbue()`、`copyfmt()` 或者 `~ios_base()` 时，就会分别调用为 `imbue_event`、`copyfmt_event` 或 `erase_event` “注册”的函数。在状态改变时，将以注册时提供给 `register_callback()` 的参数 `i` 去调用那些以前注册了的函数。

这种存储和回调机制并不很清晰。请仅仅在你绝对需要扩充低级格式化机制的时候再去使用它。

21.8 C输入/输出

因为C++ 和C是交织在一起的，有时会有C++ 的流I/O和C的 `printf()` 一族I/O函数混合使用的情况。C风格的I/O函数由 `<cstdio>` 和 `<stdio.h>` 提供。还有，因为可以从C++ 里调用C函数，有些程序员可能喜欢使用自己更熟悉的C的I/O函数。这样，即使你喜欢流I/O，你也不可避免地会在某个时候遇到C风格的I/O。

C和C++ 的I/O可以在按字符工作的基础上混合使用。在程序的执行中第一次流I/O操作之前调用一次 `sync_with_stdio()`，就可以保证C风格的和C++ 风格的I/O操作能够共享缓冲区。在第一次流I/O操作之前调用 `sync_with_stdio(false)` 将阻止缓冲区共享，这样做，在有些实现中可能改进I/O的性能。

```

class ios_base {
    // ...
    static bool sync_with_stdio(bool sync = true); // 取值和设置
};

```

与C标准库函数 `printf()` 相比，流I/O输出函数的普遍优势在于这些函数是类型安全的，而且在表述内部类型和用户定义类型的对象输出时采用了一种统一的风格。

通用的C输出函数：

```

int printf(const char* format ...); // 写入stdout
int fprintf(FILE*, const char* format ...); // 写入“文件”(stdout、stderr等)
int sprintf(char* p, const char* format ...); // 写入p[0] ...

```

在格式串 `format` 的控制下，产生任意参数序列的格式化输出。格式串里包含两类对象：普通字符，它们将直接复制到输出流；转换描述，每个描述导致对下一个参数的转换和输出。每个转换描述都由 `%` 引导。例如，

```
printf("there were %d members present.", no_of_members);
```

这里的 `%d` 说明应将 `no_of_members` 作为 `int` 处理，并打印为适当的十进制数字序列。如果有 `no_of_members == 127`，输出将是

```
there were 127 members present.
```

转换描述集相当大，而且提供了极大的灵活性。在 `%` 之后可以有：

- 可选的负号, 要求将转换后的值在输出域中居左调整。
- + 可选的正号, 要求有符号类型的值总带有一个 + 或 - 符号。
- 0 可选的0, 要求用0作为数值的前导填充。如果有 - 或给出了精度则忽略0。
- # 可选的 #, 要求浮点数总带有小数点, 即使随后没有非0字符, 尾随的0也都打印; 八进制值打印时总带着0前缀, 十六进制值打印时总带着0x或0X前缀。
- d 可选的数字序列描述域的宽度; 如果转换结果的字符数少于这个域宽, 其左边用空格填充 (或者是右边填充, 如果给出了居左指示符) 以达到域宽。如果域宽描述之前有0, 则用0填充而不用空格。
- . 可选的圆点, 用于分隔域宽和随后的数字序列。
- d 可选的数字序列描述精度, 它确定在采用e或f转换时出现在小数点之后的数字个数, 以及对一个字符串打印的最大字符个数。
- * 域宽或精度可以是 * 而不是数字序列, 这时要求通过整型参数提供域宽或精度。
- h 可选字符h, 说明随后的d、o、x、u对应于短整数参数。
- l 可选字符l, 说明随后的d、o、x、u对应于长整数参数。
- % 说明应打印字符 %, 不使用参数。
- c 一个字符, 指明应该实施的转换类型。转换字符及其含义如下:
 - d 将整数参数转换为十进制表示。
 - i 将整数参数转换为十进制表示。
 - o 将整数参数转换为八进制表示。
 - x 将整数参数转换为十六进制表示。
 - X 将整数参数转换为十六进制表示。
 - f float或double参数转换为 [-]ddd.ddd风格的十进制表示。小数点后的d的个数等于参数的精度描述, 如果需要则做舍入。如果未提供精度则采用6位精度, 如果明确给出精度0且没有#, 那么就不打印小数点。
 - e float或double参数转换为 [-]d.ddde+dd或 [-]d.ddde-dd的科学表示形式, 其中小数点前一位数字, 小数点后的数字个数等于对参数的精度描述, 如果需要则做舍入。如果未提供精度则采用6位精度, 如果明确给出精度0且没有#, 那么就不打印小数点。
 - E 与e相同, 只是用E作为指数的指示符。
 - g float或double参数用d风格、f风格或者e风格打印, 看哪种形式能在最小的空间中给出最大的精度。
 - G 与g相同, 只是用E作为指数的指示符。
 - c 打印字符参数, 如遇空字符则忽略。
 - s 参数取字符串 (字符指针), 打印串中字符, 直至遇到空字符或者达到由精度描述指明的字符个数; 如果没给精度或给0, 就打印出直至空字符的所有字符。
 - p 参数应为指针, 打印出的形式由实现确定。
 - u 无符号整数转换为十进制表示。
 - n 在调用printf()、sprintf()或fprintf()时至此写出去的字符数, 这一数值被写进由int指针参数所指的int里。

不存在域宽或小的域宽都不会导致域内容截断的情况, 如果域宽超过实际宽度则进行填

充。下面是一些更详细的例子：

```
char* line_format = "#line %d \"%s\"\\n";
int line = 13;
char* file_name = "C++/main.c";

printf("int a;\\n");
printf(line_format, line, file_name);
printf("int b;\\n");
```

这将产生

```
int a;
#line 13 "C++/main.c"
int b;
```

用`printf()`在某种意义上说是不安全的，因为它不做类型检查。举例来说，下面是一种众所周知的得到不可预见输出、内核卸载或者更糟糕情况的方法：

```
char x;
// ...
printf("bad input char: %s", x);           // %s 应该是 %c
```

当然，`printf()`确实以一种C程序员都熟悉的形式提供了极大的灵活性。

与此类似，`getchar()`提供了一种从输入读字符的习惯方式：

```
int i;
while ((i=getchar()) != EOF) { /* 使用i */ }
```

注意，为能对照`int`值`EOF`检测文件结束，必须将`getchar()`的值放入一个`int`而不是`char`。

关于C语言I/O的进一步细节，请查看你的C语言参考手册或者Kernighan和Reitche的《The C Programming Language》[Kernighan, 1988]。

21.9 忠告

- [1] 在为用户定义类型的值定义 `<<` 和 `>>` 时，应该采用意义清晰的正文表示形式；21.2.3节、21.3.5节。
- [2] 在打印包含低优先级运算符的表达式时需要使用括号；21.2节。
- [3] 在添加新的 `<<` 和 `>>` 运算符时，你不必修改`istream`或`ostream`；21.2.3节。
- [4] 你可以定义函数，使其能基于第二个（或更后面的）参数，具有像`virtual`函数那样行为；21.2.3.1节。
- [5] 切记，按默认约定 `>>` 跳过所有空格；21.3.2节。
- [6] 使用低级输入函数（如`get()`和`read()`）主要是为了实现高级输入函数；21.3.4节。
- [7] 在使用`get()`、`getline()`和`read()`时留心其终止准则；21.3.4节。
- [8] 在控制I/O时，尽量采用操控符，少用状态标志；21.3.3节、21.4节、21.4.6节。
- [9] （只）用异常去捕捉罕见的I/O错误；21.3.6节。
- [10] 联结用于交互式I/O的流；21.3.7节。
- [11] 使用哨位将许多函数的入口和出口代码集中到一个地方；21.3.8节。
- [12] 在无参数操控符最后不要写括号；21.4.6.2节。
- [13] 使用标准操控符时应记住写 `#include <iomanip>`；21.4.6.2节。
- [14] 你可以通过定义一个简单函数对象得到三元运算符的效果（和效率）；21.4.6.3节。

- [15] 切记, *width* 描述只应用于随后的一个I/O操作; 21.4.4节。
- [16] 切记*precision*描述只对随后所有的浮点数输出操作有效; 21.4.3节。
- [17] 用字符串流做内存里的格式化; 21.5.3节。
- [18] 你可以描述一个文件流的模式; 21.5.1节。
- [19] 在扩充I/O系统时, 应该清楚地区分格式化 (*iostream*) 和缓冲 (*streambuf*); 21.1节、21.6节。
- [20] 将传输值的非标准方式实现为流缓冲区; 21.6.4节。
- [21] 将格式化值的非标准方式实现为流操作; 21.2.3节、21.3.5节。
- [22] 你可以利用一对函数隔离和封装起对用户定义代码的调用; 21.6.4节。
- [23] 你可以在读入之前用*in_avail()* 去确定输入操作是否会被阻塞; 21.6.4节。
- [24] 划分清楚需要高效的简单操作和实现某种策略的操作 (将前者做成*inline*, 将后者做成*virtual*); 21.6.4节。
- [25] 用*locale*将“文化差异”局部化; 21.7节。
- [26] 用*sync_with_stdio(x)* 去混合C风格和C++ 风格的I/O, 或者离解C风格和C++ 风格的I/O; 21.8节。
- [27] 当心C风格I/O的类型错误; 21.8节。

21.10 练习

1. (*1.5) 读入一个浮点数文件, 由其中的每一对数做出一个复数, 并输出这些复数。
2. (*1.5) 定义类型*Name_and_address*, 为它定义 << 和 >>。复制一个*Name_and_address*对象的流。
3. (*2.5) 复制一个*Name_and_address*对象的流, 你应在其中加入可以想到的尽可能多的错误 (例如, 格式错和字符串提前结束等)。采用某种方式处理这些错误, 并保证即使输入完全乱作一团, 复制函数也能够读入最多的格式正确的*Name_and_address*。
4. (*2.5) 重新定义*Name_and_address*的I/O格式, 使之能更好地抵御遇到的格式错误。
5. (*2.5) 设计一些要求并读入各种类型信息的函数。想法: 整数、浮点数、文件名、通信地址、日期、个人信息等。设法使它们在任何情况下都能保证不出错。
6. (*1.5) 写一个程序打印: (a) 所有小写字母, (b) 所有字母, (c) 所有字母和数字, (d) 所有在你的系统里可以出现在C++ 标识符中的字符, (e) 所有标点符号字符, (f) 所有控制字符的整数值, (g) 所有空白字符, (h) 所有空白字符的整数值, (i) 所有可打印字符。
7. (*2) 将一系列正文行读入一个固定大小的字符缓冲区。删除所有空白字符, 将所有字母数字字符用字母表里的下一个字符取代 (*z*用*a*取代, *9*用*0*取代), 而后写出取代之后的结果。
8. (*3) 写一个“微缩本”的流I/O系统, 它提供了类*istream*、*ostream*、*ifstream*、*ofstream*, 它们提供了一些函数 (如对整数的*operator<<()* 和*operator>>()*) 以及对文件的操作 (如*open()* 和*close()* 等)。
9. (*4) 用C++ 标准I/O库 (<*iostream*>) 实现C标准I/O库 (<*stdio.h*>)。
10. (*4) 用C标准I/O库 (<*stdio.h*>) 实现C++ 标准I/O库 (<*iostream*>)。
11. (*4) 实现C和C++ 库, 使之能够同时使用。
12. (*2) 实现一个类, 它的 [] 被重载, 以实现从文件中字符的随机访问。

原书缺页

第22章 数 值

计算的意义在于洞察力，
而不是数字。

—— R.W. Hamming

……但对学生而言，
数字常常是取得洞察力的最好途径。

—— A. Ralston

引言——数值限制——数学函数——*valarray*——向量操作——切割——*slice_array*
——临时量删除——*gslice_array*——*mask_array*——*indirect_array*——*complex*
——通用算法——随机数——忠告——练习

22.1 引言

什么计算都不做的实际代码是很罕见的。当然，大部分代码都需要一些超出简单算术的数学。本章将展示标准库为在这一方向上走得更远的人们提供的各种功能。

C和C++的基本设计都没有将数值计算放在心里。然而，数值计算也常常出现在其他工作的情景之中，例如数据访问、网络、仪器控制、图形、模拟、金融分析等等，所以，C++也就变成了一种有吸引力的媒介，用于实现作为大系统中一部分的计算。进一步说，数值方法早已远远超越了在浮点数向量上的简单循环。在那些需要用更复杂的数据结构作为计算中一些部分的地方，C++的威力就能发挥作用了。这些因素的综合影响使C++越来越多地被应用于涉及复杂数值的科学与工程计算。由此，支持这类计算的功能和技术也逐渐浮现出来。本章将介绍标准库里支持数值的部分，并展示一些技术以处理人们在用C++表述数值计算时引出的问题。我并不企图去教数值方法。数值计算本身也是一个迷人的领域，为理解它，你需要上一门很好的数值方法课或者至少需要一本好教材——而不止是一本语言手册或简单讲义。

22.2 数值的限制

要想对数值做任何有趣的事情，我们常常需要对于内部数值类型的某些一般性质有所了解。这些性质是由实现定义的，而不是由语言本身的规则固定的（4.6节）。例如，什么是最大的`int`？什么是最小的`float`？用`double`给`float`赋值时做的是舍入还是截断？在一个`char`里有多少个二进制位？

对于这类问题的回答都由`numeric_limits`模板的专门化提供，它们位于`<limits>`之中。例如，

```
void f(double d, int i)
{
```

```

    if (numeric_limits<unsigned char>::digits != 8) {
        // 罕见的字节（不是8位）
    }

    if (i < numeric_limits<short>::min() || numeric_limits<short>::max() < i) {
        // i不可能存入short而又不丢掉主要部分
    }

    if (0 < d && d < numeric_limits<double>::epsilon()) d = 0;

    if (numeric_limits<Quad>::is_specialized) {
        // 类型Quad的限制信息可用
    }
}

```

每个专门化都为其参数类型提供了有关的信息。这样，最一般的**numeric_limits**模板也就是对一组常量和在线函数的记法句柄：

```

template<class T> class numeric_limits {
public:
    static const bool is_specialized = false; // numeric_limits<T> 的限制信息可用吗?

    // 没意思的默认值
};

```

所有实际信息都在各个专门化里。每个标准库实现都为各个数值类型（各个字符类型、整数类型和浮点数类型，以及**bool**类型）提供了**numeric_limits**的专门化版本，但没有为任何其他貌似候选者的类型提供这种描述，例如**void**、枚举或者库类型（如**complex <double>**）。

对于一个像**char**这样的整型，有些意思的信息并不多。这里给出的是某个实现里的**numeric_limits<char>**，其中的**char**是8位且有符号：

```

class numeric_limits<char> {
public:
    static const bool is_specialized = true; // 是，这里有信息

    static const int digits = 7; // 二进制位（二进制数字）个数，不包括符号位

    static const bool is_signed = true; // 这个实现采用有符号字符
    static const bool is_integer = true; // char是一个整型

    static char min() throw() { return -128; } // 最小值
    static char max() throw() { return 127; } // 最大值

    // 大量与char无关的声明
};

```

注意，对于有符号整型，**digits**比用于存储这个类型的二进制位数少1。

numeric_limits模板的大部分成员都是为了描述浮点数。例如，下面是某个可能实现里的**float**的描述：

```

class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static const int radix = 2; // 指数的基数（这里采用2）
    static const int digits = 24; // 尾数按照基数的位数
    static const int digits10 = 6; // 尾数按照十进制的位数

    static const bool is_signed = true;
    static const bool is_integer = false;
};

```

```

static const bool is_exact = false;

static float min() throw() { return 1.17549435E-38F; }
static float max() throw() { return 3.40282347E+38F; }

static float epsilon() throw() { return 1.19209290E-07F; }
static float round_error() throw() { return 0.5F; }

static float infinity() throw() { return /* 某个值 */; }
static float quiet_NaN() throw() { return /* 某个值 */; }
static float signaling_NaN() throw() { return /* 某个值 */; }
static float denorm_min() throw() { return min(); }

static const int min_exponent = -125;
static const int min_exponent10 = -37;
static const int max_exponent = +128;
static const int max_exponent10 = +38;

static const bool has_infinity = true;
static const bool has_quiet_NaN = true;
static const bool has_signaling_NaN = true;
static const float denorm_style has_denorm = denorm_absent; // 来自 <limits> 的enum
static const bool has_denorm_loss = false;

static const bool is_iec559 = true; // 符合IEC-559
static const bool is_bounded = true;
static const bool is_modulo = false;
static const bool traps = true;
static const bool tinyness_before = true;

static const float_round_style round_style = round_to_nearest; // 来自 <limits> 里的enum
};

```

注意, `min()` 是最小的正的正规化数, 而 `epsilon` 是最小的那个正的且使 $1 + \text{epsilon} - 1$ 可以被表示的浮点数。

在按照内部类型的方式定义标量类型时, 同时提供一个适宜的 `numeric_limits` 的专门化是一个很好的方法。举例说, 如果我写了一个四精度类型 `Quad`, 或者某个开发商提供了一个扩展精度整数 `long long`, 用户期望有 `numeric_limits<Quad>` 和 `numeric_limits<long long>` 也是很合理的。

人们也可以设想用 `numeric_limits` 的专门化去描述与浮点数毫无关系的用户定义类型的性质。一般在这种情况下, 更好的方式是采用通用的技术去描述该类型的性质, 而不是去专门化 `numeric_limits`, 使之包含一些标准中未考虑的性质。

`numeric_limits` 里的浮点值都被表示成在线函数, 但这里的整型值必须用某种形式表示, 以保证能够用在常量表达式里。这就意味着它们必须有在类里的初始式 (10.4.6.2 节)。如果你采用 `static const` 成员而不是枚举符, 那就必须记住去定义那些 `static`。

22.2.1 表示限制的宏

C++ 从 C 继承了描述整数性质的宏, 这些都可以在 `<climits>` 和 `<limits.h>` 里找到, 名字如 `CHAR_BIT` 和 `INT_MAX` 等。类似地, 在 `<cfloat>` 和 `<float.h>` 里定义了描述浮点数性质的宏, 它们的名字如 `DBL_MIN_EXP`、`FLT_RADIX` 和 `LDBL_MAX` 等。

与其他地方一样, 最好是避免使用宏。

22.3 标准数学函数

头文件 `<cmath>` 和 `<math.h>` 提供了通常所说的“常用数学函数”:


```

double abs(double);           // 绝对值, 不在C中, 同fabs()
double fabs(double);          // 绝对值

double ceil(double d);        // 不小于d的最小整数
double floor(double d);       // 不大于d的最大整数

double sqrt(double d);        // d的平方根, d必须非负

double pow(double d, double e); // d的e次幂;
                                // 如果d == 0 且 e <= 0 或者 d < 0 且 e 不
double pow(double d, int i);   // 是整数, 则错误d的i次幂; 不是在c中

double cos(double);           // 余弦
double sin(double);           // 正弦
double tan(double);           // 正切

double acos(double);          // 反余弦
double asin(double);          // 反正弦
double atan(double);          // 反正切
double atan2(double x, double y); // atan(x/y)

double sinh(double);          // 双曲正弦
double cosh(double);          // 双曲余弦
double tanh(double);          // 双曲正切

double exp(double);           // 指数, 以e为底
double log(double d);         // 自然对数 (以e为底), d必须大于0
double log10(double d);       // 10底对数, d必须大于0

double modf(double d, double* p); // 返回d的小数部分, 整数部分存入 *p
double frexp(double d, int* p);   // 找出 [0.5, 1) 中的x和y, 使d = x *
                                // pow(2, y), 返回x并将y存入 *p

double fmod(double d, double m); // 浮点数余数, 符号与d相同
double ldexp(double d, int i);    // d * pow(2, i)

```

此外, `<cmath>` 和 `<math.h>` 还为 `float` 和 `long double` 参数提供了这些函数。

在可能有多个结果值的地方 (例如 `asin()`), 就返回距0最近的那个结果。`acos()` 的结果总为非负。

错误报告的方式是设置来自 `<errno>` 的 `errno`, 在出现作用域错误时将它设为 `EDOM`, 出现值域错误时将它设置为 `ERANGE`。例如,

```

void f()
{
    errno = 0; // 清除errno原值
    sqrt(-1);
    if (errno == EDOM) cerr << "sqrt() not defined for negative argument";
    pow(numeric_limits<double>::max(), 2);
    if (errno == ERANGE) cerr << "result of pow() too large to represent as a double";
}

```

由于历史原因, 有几个数学函数位于 `<cstdlib>` 里而不是 `<cmath>` 里:

```

int abs(int);           // 绝对值
long abs(long);         // 绝对值 (不在C中)
long labs(long);        // 绝对值

struct div_t { implementation_defined quot, rem; };
struct ldiv_t { implementation_defined quot, rem; };

div_t div(int n, int d); // 用d除n, 返回 (商, 余数)
ldiv_t div(long int n, long int d); // 用d除n, 返回 (商, 余数) (不在C中)
ldiv_t ldiv(long int n, long int d); // 用d除n, 返回 (商, 余数)

```

22.4 向量算术

许多数值工作都依赖于相对简单的浮点数值的一维向量。特别是，这种向量得到了一些高性能机器的体系结构的很好支持，基于这种向量的库得到广泛使用，采用这种向量的许多极具进取心的优化代码在许多领域中都被认为是必不可少的。因此，标准库也提供了一种向量，称为 *valarray*，其特殊设计就是为了加速常用数值向量操作。

在观察 *valarray* 的功能时，应该记住它们的意图就是作为高性能计算的一种相对低级的构件。特别要注意，*valarray* 的基本设计准则并不是易用性，而是在极度优化的代码的基础上，更有效地利用高性能计算机。如果你的目标是灵活性和通用性而不是效率，那么最好是在第16章和第17章的标准容器基础之上搞建设，而不是去适应 *valarray* 简单、高效、精致脆弱的传统框架。

有人可能争辩说，*valarray* 原本就该叫做 *vector*，因为它就是传统的数学上的向量，而 *vector*（16.3节）则应该称为 *array*。无论如何，这里并不是讨论术语的地方。*valarray* 就是一种特别为数值计算优化了的向量，*vector* 则是一种灵活的容器，其设计就是为了保存和操控类型广泛多样的对象，而数组则是一种低级的内部类型。

valarray 类型是由4个用于刻画 *valarray* 中各个部分的辅助类型支持的：

- *slice_array* 和 *gslice_array* 表述切割的概念（22.4.6节、22.4.8节）。
- *mask_array* 通过屏蔽各个元素的进和出来刻画数据的子集（22.4.9节）。
- *indirect_array* 列出需要考虑的元素的下标（22.4.10节）。

22.4.1 *valarray* 的构造

valarray 类型及其相关功能定义在名字空间 *std* 里，由 `<valarray>` 给出：

```
template<class T> class std::valarray {
    // 表示
public:
    typedef T value_type;

    valarray(); // valarray, size() == 0
    explicit valarray(size_t n); // n 个元素，值都为 T()
    valarray(const T& val, size_t n); // n 个元素，值都为 val
    valarray(const T* p, size_t n); // n 个元素，值为 p[0], p[1], ...
    valarray(const valarray& v); // 复制 v

    valarray(const slice_array<T>&); // 见22.4.6节
    valarray(const gslice_array<T>&); // 见22.4.8节
    valarray(const mask_array<T>&); // 见22.4.9节
    valarray(const indirect_array<T>&); // 见22.4.10节

    ~valarray();

    // ...
};
```

这一组构造函数使我们可以用辅助的数值数组或者单个的值做 *valarray* 的初始化。例如，

```
valarray<double> v0; // 占位，可在后面给 v0 赋值
valarray<float> v1(1000); // 1000 个元素，值都是 float() == 0.0F

valarray<int> v2(-1, 2000); // 2000 个元素，值为 -1
valarray<double> v3(100, 9.8064); // 大错：浮点值 valarray 大小

valarray<double> v4 = v3; // v4 具有 v3.size() 个元素
```

对那些两个参数的构造函数，元素值都出现在元素个数之前。这一点也与其他标准容器不同（16.3.4节）。

提供给复制构造函数的参数`valarray`的元素个数就确定了作为结果的`valarray`的大小。

大部分程序都需要来自数据表或者输入的数据，这由一个从内部数组复制元素的构造函数支持。例如，

```
const double vd[] = { 0, 1, 2, 3, 4 };
const int vi[] = { 0, 1, 2, 3, 4 };

valarray<double> v3(vd, 4);    // 4个元素: 0, 1, 2, 3
valarray<double> v4(vi, 4);    // 类型错: vi不是double指针
valarray<double> v5(vd, 8);    // 无定义: 初始式元素太少
```

这种初始化形式很重要，因为数值软件常常以大数组的形式产生数据。

`valarray`及其辅助功能都是为高速计算设计的，这种情况也反应在对用户的若干约束和为库实现者做出的某些保证方面。简而言之，`valarray`的实现者能够使用自己所知道的任何优化技术，例如，操作可以在线化，对`valarray`的操作可以假设都无副作用（当然，除了对于它们的显式参数）。还有，`valarray`假定为无别名的，在维持基本语义的前提下，允许引进辅助类型或删除各种临时量。这样，在`<valarray>`里的声明看起来可能与你将在这里看到的（或者在标准里看到的）有些不同，但它们应该是提供了同样的操作，其代码具有同样的语义，不会出格到破坏了这些基本规则。特别值得指出的是，`valarray`的元素应该使用普通的复制语义（17.1.4节）。

22.4.2 `valarray`的下标和赋值

对于`valarray`，下标操作同时用于访问元素和获取子数组：

```
template<class T> class valarray {
public:
    // ...
    valarray& operator=(const valarray& v);    // 复制
    valarray& operator=(const T& val);         // 对每个元素用val赋值

    T operator[] (size_t) const;
    T& operator[] (size_t);

    valarray operator[] (slice) const;         // 见22.4.6节
    slice_array<T> operator[] (slice);

    valarray operator[] (const gslice&) const; // 见22.4.8节
    gslice_array<T> operator[] (const gslice&);

    valarray operator[] (const valarray<bool>&) const; // 见22.4.9节
    mask_array<T> operator[] (const valarray<bool>&);

    valarray operator[] (const valarray<size_t>&) const; // 见22.4.10节
    indirect_array<T> operator[] (const valarray<size_t>&);

    valarray& operator=(const slice_array<T>&); // 见22.4.6节
    valarray& operator=(const gslice_array<T>&); // 见22.4.8节
    valarray& operator=(const mask_array<T>&); // 见22.4.9节
    valarray& operator=(const indirect_array<T>&); // 见22.4.10节

    // ...
};
```

可以将一个 *valarray* 赋给另一个同样大小的 *valarray*。正如我们所期望的，*v1 = v2* 将 *v2* 的每个元素都赋给 *v1* 的对应位置。如果 *valarray* 大小不同，赋值的结果将无定义。因为 *valarray* 设计时对速度做了优化，假定采用大小不正确的 *valarray* 赋值会导致某种易理解的错误（如抛出某个异常）或者完成某种“合理的”动作，这些想法都很不明智。

除了常规赋值之外，也可能用标量给 *valarray* 赋值。例如，*v = 7* 将 7 赋值给 *v* 的每个元素。这可能令人感到吃惊，但最好是将它理解为赋值运算符操作的一种偶尔非常有用的退化情况（22.4.3 节）。

用整数的下标具有常规的行为，且不做范围检查。

除了选择个别的元素之外，*valarray* 的下标操作还提供了四种方式，以便能提取子数组（22.4.6 节）。与此相对应，赋值（和构造函数）也能接受子数组作为运算对象。在 *valarray* 上的赋值运算集合都保证，不必将如 *slice_array* 等各种辅助数组类型转换到 *valarray* 就可以直接赋值。实现也可以采用类似方式，为了保证效率而提供其他向量运算的多份不同版本，如 *+* 和 ***。此外，还存在着许多威力强大的优化技术，可用于涉及切割和其他辅助向量类型的向量操作。

22.4.3 成员操作

这里提供了所有最明显的和若干不那么明显的成员函数：

```
template<class T> class valarray {
public:
    // ...
    valarray& operator*=(const T& arg);    // v[i] *= arg, 对每个元素
    // 类似地: /=, %=, +=, -=, ^=, &=, |=, <=< 和 >>=

    T sum() const;                        // 元素求和, 用 += 做加法
    T min() const;                        // 最小值, 用 < 比较
    T max() const;                        // 最大值, 用 < 比较

    valarray shift(int i) const;          // 逻辑移位 (0<i左移, i<0右移)
    valarray cshift(int i) const;        // 循环移位 (0<i左移, i<0右移)

    valarray apply(T f(T)) const;        // 对每个元素, result[i] = f(v[i])
    valarray apply(T f(const T&)) const;

    valarray operator-() const;          // 对每个元素, result[i] = -v[i]
    // 类似地: +, ~, .

    size_t size() const;                  // 元素个数
    void resize(size_t n, const T& val = T()); // n个值为val的元素
};
```

如果 *size() == 0*，*sum()*、*min()* 和 *max()* 的值无定义。

例如，如果 *v* 是一个 *valarray*，它就可以用 *v *= .2* 和 *v /= 1.3* 改变尺度。也就是说，将一个标量作用于一个向量意味着将这个标量作用于向量的每个元素。如常，对使用 **=* 的优化要比对使用 *** 和 *=* 的优化更容易（11.3.1 节）。

注意，非赋值操作总是创建一个新的 *valarray*。例如，

```
double incr(double d) { return d+1; }

void f(valarray<double>& v)
{
    valarray<double> v2 = v.apply(incr);    // 生成增量后的valarray
}
```

这并没有改变`v`的值。不幸的是, `apply()` 不能接受函数对象 (18.4节) 作为参数 (22.9[1])。

逻辑和循环移位函数 `shift()` 和 `cshift()` 返回一个元素适当移动后得到的新的 `valarray`, 且原数组不变。例如, 循环移位 `v2 = v.cshift(n)` 生成一个 `valarray`, 使 `v2[i] = v[(i + n) % v.size()]`。逻辑移位 `v3 = v.shift(n)` 生成一个 `valarray`, 使在 `i + n` 是 `v` 的合法下标时, `v3[i]` 等于 `v[i + n]`; 否则 `v[i]` 取默认元素值。这也意味着在参数为正时, `shift()` 和 `cshift()` 完成左移, 参数为负时做右移。例如,

```
void f()
{
    int alpha[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    valarray<int> v(alpha, 8);           // 1, 2, 3, 4, 5, 6, 7, 8
    valarray<int> v2 = v.shift(2);       // 3, 4, 5, 6, 7, 8, 0, 0
    valarray<int> v3 = v<<2;             // 4, 8, 12, 16, 20, 24, 28, 32
    valarray<int> v4 = v.shift(-2);      // 0, 0, 1, 2, 3, 4, 5, 6
    valarray<int> v5 = v>>2;             // 0, 0, 0, 1, 1, 1, 1, 2
    valarray<int> v6 = v.cshift(2);     // 3, 4, 5, 6, 7, 8, 1, 2
    valarray<int> v7 = v.cshift(-2);    // 7, 8, 1, 2, 3, 4, 5, 6
}
```

对于 `valarray`, `>>` 和 `<<` 都是二进制位移运算符, 而不是元素移位运算符, 也不是 I/O 运算符 (22.4.4节)。因此, 可以用 `<<=` 和 `>>=` 在整型 `valarray` 的元素内部移位。例如,

```
void f(valarray<int> vi, valarray<double> vd)
{
    vi <<= 2; // 对vi的所有元素做vi[i] <<= 2
    vd <<= 2; // 错误: 浮点值移位无定义
}
```

改变 `valarray` 的大小是可能的。然而, 提供 `resize()` 操作并不是想将 `valarray` 做成一个数据结构, 使它能像 `vector` 或者 `string` 那样动态增长。相反, `resize()` 是一个重新初始化的操作, 它将一个 `valarray` 的现存内容用一组默认值取代, 把原有的值都丢掉。

通常, 一个重新设置大小的 `valarray` 就是我们想建的一个空向量。考虑我们可能怎样用输入来初始化一个 `valarray`:

```
void f()
{
    int n = 0;
    cin >> n;                               // 读入数组大小
    if (n <= 0) error("bad array bound");

    valarray<double> v(n);                   // 做出正确大小的数组
    int i = 0;
    while (i < n && cin >> v[i++]);          // 填充数组
    if (i != n) error("too few elements on input");

    // ...
}
```

如果我们想用一个单独的函数处理输入, 那么就可以像下面这样做:

```
void initialize_from_input(valarray<double>& v)
{
    int n = 0;
    cin >> n;                               // 读入数组大小
    if (n <= 0) error("bad array bound");
```

```

    v.resize(n); // 做出正确大小的数组
    int i = 0;
    while (i < n && cin >> v[i++]); // 填充数组
    if (i != n) error("too few elements on input");
}

void g()
{
    valarray<double> v; // 做一个默认数组
    initialize_from_input(v); // 给v确定正确的大小和元素

    // ...
}

```

这样就可以避免复制大量的数据。

如果我们想让一个保存着有价值数据的`valarray`动态增长，那就必须用一个临时向量：

```

void grow(valarray<int>& v, size_t n)
{
    if (n <= v.size()) return;

    valarray<int> tmp(n); // n个默认元素

    copy(&v[0], &v[v.size()], &tmp[0]); // 用18.6.1节的复制算法
    v.resize(n);
    copy(&tmp[0], &tmp[v.size()], &v[0]);
}

```

这并不是使用`valarray`的理想方式。`valarray`的意图是在给定初始值之后保持大小不变。

`valarray`的元素形成了一个序列，也就是说，`v[0] .. v[n-1]` 在内存中连续排列。这也意味着，对于`valarray<T>`的`T*` 是一个随机访问迭代器，所以各种标准算法（例如`copy()`等）都能对它使用。然而，用赋值和子数组的方式表述复制才更符合`valarray`的精神：

```

void grow2(valarray<int>& v, size_t n)
{
    if (n <= v.size()) return;

    valarray<int> tmp = v;
    slice s(0, v.size(), 1); // v.size() 个元素的子数组（见22.4.5节）

    v.resize(n); // 修改大小不能保持元素值
    v[s] = tmp; // 将元素复制回v的前面部分
}

```

如果因为某些原因，输入数据的组织使你必须统计出元素个数，而后才能知道保存它们所需要的向量的大小，那么最好是先将输入读进一个`vector`（16.3.5节），而后再将它们复制到`valarray`里。

22.4.4 非成员函数

这里提供了常用的二元运算符和数学函数：

```

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);

// 类似地：/, %, +, -, ^, &, |, <<, >>, &&, ||, ==, !=, <, >, <=, >=, atan2 和 pow

template<class T> valarray<T> abs(const valarray<T>&);

// 类似地：acos, asin, atan, cos, cosh, exp, log, log10, sin, sinh, sqrt, tan 和 tanh

```

这些二元运算符都是为`valarray`以及`valarray`与其标量类型的组合而定义的。例如,

```
void f(valarray<double>& v, valarray<double>& v2, double d)
{
    valarray<double> v3 = v*v2;    // 对所有的i, v3[i] = v[i]*v2[i]
    valarray<double> v4 = v*d;    // 对所有的i, v4[i] = v[i]*d
    valarray<double> v5 = d*v2;    // 对所有的i, v5[i] = d*v2[i]

    valarray<double> v6 = cos(v); // 对所有的i, v6[i] = cos(v[i])
}
```

像上面例子中的`*`和`cos()`所表明的那样, 这些向量运算符都将它们的操作应用到其运算对象的每一个成员上。自然, 一个操作能够应用的条件是对应的模板参数类型也定义有相应的操作; 否则编译器在设法做模板实例化时就会发生错误(13.5节)。

如果运算结果也是`valarray`, 它的长度将与运算对象的长度相同。如果两个数组长度不同, 对于这两个`valarray`使用二元运算的结果没有定义。

很奇怪, 对于`valarray`并没有提供I/O操作(22.4.3节), `<<`和`>>`是移位运算符。当然, 很容易定义出针对`valarray`的`<<`和`>>`的I/O版本(22.9[5])。

注意, 这些`valarray`运算将返回新的`valarray`, 而不是修改它们的运算对象。这样做可能代价高昂, 但如果要使用高度的优化技术, 就必须这样做(例如, 见22.4.7节)。

所有针对`valarray`的运算符和数学函数都可以用于`slice_array`(22.4.6节)、`gslice_array`(22.4.8节)、`mask_array`(22.4.9节)、`indirect_array`(22.4.10节)以及这些类型的组合。当然, 具体实现也可以先将非`valarray`转换为`valarray`, 而后再执行所需运算。

22.4.5 切割

一个`slice`(切割)是一个抽象, 它使我们能有效地将一个向量当做任意维的数组去操作。切割也是Fortran向量和BLAS(Basic Linear Algebra Subprogramms)库的基本概念, 这个库是许多数值计算的基础。简而言之, 一个切割就是在一个`valarray`里某一部分中间距为`n`的一系列元素:

```
class std::slice {
    // 开始下标、长度和跨步
public:
    slice();
    slice(size_t start, size_t size, size_t stride);

    size_t start() const;    // 第一个元素的下标
    size_t size() const;    // 元素个数
    size_t stride() const;   // 元素n位于start()+n*stride()
};
```

`stride`(跨步)是`slice`里的两个元素之间的距离(元素个数)。这样, 一个`slice`就描述了一个整数序列。例如,

```
size_t slice_index(const slice& s, size_t i) // 将i映射到它对应的下标
{
    return s.start()+i*s.stride();
}

void print_seq(const slice& s) // 打印出s的元素
{
    for (int i = 0; i < s.size(); i++) cout << slice_index(s, i) << " ";
}
```

```

}
void f()
{
    print_seq(slice(0, 3, 4)); // 0行
    cout << " ";
    print_seq(slice(1, 3, 4)); // 1行
    cout << " ";
    print_seq(slice(0, 4, 1)); // 0列
    cout << " ";
    print_seq(slice(4, 4, 1)); // 1列
}

```

将打印出0 4 8, 1 5 9, 0 1 2 3, 4 5 6 7。

换句话说, 一个slice描述了从整数到下标的一个映射。元素的数目(`size()`)并不影响这个映射(定址), 而只是使我们能找到序列的终点。这种映射提供了一种高效、通用和合理的方式, 使我们能在一个一维数组(例如`valarray`)里模拟一个二维数组。现在, 按照我们通常所设想的方式考虑一个3乘4矩阵(C.7节):

00	01	02
10	11	12
20	21	22
30	31	32

按照Fortran的规定, 这个矩阵在内存里的布局如下:

0				4				8			
00	10	20	30	01	11	21	31	02	12	22	32
0	1	2	3								

这并不是在C++里数组布局的方式(见C.7节); 然而, 我们应该能给出一个概念, 它具有清晰的符合逻辑的界面, 而后选择一种表示方式去适应这个问题的约束条件。在这里我选用了Fortran的布局, 这样做就是为了能很容易地与遵循这种约定的数值软件交换信息。当然, 我并没有走得足够远, 例如, 让下标从1开始而不是从0开始, 这个问题留做练习(22.9[9])。许多数值计算工作是在混合的语言里, 采用多种多样的库做的, 人们还会继续这样做。在许多情况下, 这些库和语言标准所确定数据格式多种多样, 能够操作这样的数据的能力是至关重要的。

第 x 行可以用`slice(x, 3, 4)`描述。也就是说, 第 x 行的第一个元素就是向量的第 x 个元素; 该行的下一个元素是向量的第 $x + 4$ 个元素, 如此等等。在这里每行有3个元素。在前面的图中, `slice(0, 3, 4)`描述的行是00, 01和02。

第 y 列可以用`slice(4 * y, 4, 1)`描述。也就是说, 第 y 列的第一个元素是向量的第 $4 * y$ 个元素; 该列的下一个元素是向量的第 $4 * y + 1$ 个元素, 依次类推。每列有4个元素。在前面的图中, `slice(0, 4, 1)`描述的列是00, 10, 20和30。

除了用于模拟二维数组外, `slice`还可以用于描述许多其他序列, 这是一种相当通用的刻画简单序列的方法。这一概念将在22.4.8节里进一步展开。

想像`slice`的一种方式是把它看成一种很奇特的迭代器: 一个`slice`使我们可以描述`valarray`的一个下标的序列。我们可以基于此创建一个真正的迭代器:


```

template<class T> class Slice_iter {
    valarray<T>* v;
    slice s;
    size_t curr; // 当前元素的下标

    T& ref(size_t i) const { return (*v)[s.start()+i*s.stride()]; }
public:
    Slice_iter(valarray<T>* vv, slice ss) : v(vv), s(ss), curr(0) {}

    Slice_iter end()
    {
        Slice_iter t = *this;
        t.curr = s.size(); // 末端加一元素的下标
        return t;
    }

    Slice_iter& operator++() { curr++; return *this; }
    Slice_iter operator++(int) { Slice_iter t = *this; curr++; return t; }

    T& operator[] (size_t i) { return ref(curr+i); } // C风格的下标
    T& operator() (size_t i) { return ref(curr+i); } // Fortran风格的下标
    T& operator* () { return ref(curr); } // 当前元素

    // ...
};

```

因为`slice`也有一个大小，我们甚至还可以提供范围检查。在这里我就利用`slice::size()`提供了一个`end()`操作，用它提供`valarray`里末端加一元素的迭代器。

因为`slice`既能表述行又能表述列，`Slice_iter`就使我们能够按行或者按列去遍历一个`valarray`。

为使`Slice_iter`更有用，必须将`==`、`!=`和`<`定义为友元函数：

```

template<class T> bool operator==(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr==q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}

template<class T> bool operator!=(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return !(p==q);
}

template<class T> bool operator<(const Slice_iter<T>& p, const Slice_iter<T>& q)
{
    return p.curr<q.curr && p.s.stride()==q.s.stride() && p.s.start()==q.s.start();
}

```

22.4.6 切割数组——`slice_array`

从一个`valarray`和一个`slice`，我们可以构造出某种感觉上像`valarray`，但实际上就是一种引用由切割描述的数组子集的方式。这样的—个`slice_array`可以定义如下：

```

template <class T> class std::slice_array {
public:
    typedef T value_type;

    void operator=(const valarray<T>&);
    void operator=(const T& val); // 将val赋值给每个元素
};

```

```

void operator*=(const valarray<T>& val);           // 对每个元素做v[i] *= val[i]
// 类似地: /=, %=, +=, -=, ^=, &=, |=, <<=, >>=

~slice_array();
private:
    slice_array();                               // 防止构造
    slice_array(const slice_array&);             // 防止复制
    slice_array& operator=(const slice_array&);   // 防止复制

    valarray<T>* p;                             // 实现定义表示
    slice s;
};

```

用户不能直接创建`slice_array`。实际方式是, 用户描述`valarray`的下标, 为给定切割建立起一个`slice_array`。一旦`slice_array`被初始化后, 对它的所有引用都将间接地导向创建它的那个`valarray`。例如, 我们可以如下创建某种表示数组里相间元素的东西:

```

void f(valarray<double>& d)
{
    slice_array<double>& v_even = d[slice(0, d.size()/2+d.size()%2, 2)];
    slice_array<double>& v_odd = d[slice(1, d.size()/2, 2)];

    v_odd *= 2;           // d中各奇数下标的元素加倍
    v_even = 0;           // d中各偶数下标的元素置0
}

```

禁止复制`slice_array`也是必须的, 这样才能允许做一些需要依靠无别名情况的优化。这一要求有时过于束缚人了, 例如,

```

slice_array<double> row(valarray<double>& d, int i)
{
    slice_array<double> v = d[slice(0, 2, d.size()/2)]; // 错误: 企图复制
    return d[slice(i%2, i, d.size()/2)];               // 错误: 企图复制
}

```

复制`slice`常常可以合理地代替复制`slice_array`。

可以用切割去表示数组中各种各样的子集。例如, 我们可以用如下方式去操作一些连续的子数组:

```

inline slice sub_array(size_t first, size_t count) // [first: first + count[
{
    return slice(first, count, 1);
}

void f(valarray<double>& v)
{
    size_t sz = v.size();
    if (sz < 2) return;
    size_t n = sz/2;
    size_t n2 = sz-n;

    valarray<double> half1(n);
    valarray<double> half2(n2);

    half1 = v[sub_array(0, n)]; // 复制v的前一半
    half2 = v[sub_array(n, n2)]; // 复制v的后一半

    // ...
}

```

标准库没有提供矩阵类。相反，这里的本意就是借用`valarray`和`slice`作为工具，以构造出为不同需要而优化的各种矩阵。现在来考虑我们可能如何借助于`valarray`和`slice`实现一个简单的矩阵：

```
class Matrix {
    valarray<double>* v;
    size_t d1, d2;
public:
    Matrix(size_t x, size_t y);           // 注意：没有默认构造函数
    Matrix(const Matrix&);
    Matrix& operator=(const Matrix&);
    ~Matrix();

    size_t size() const { return d1*d2; }
    size_t dim1() const { return d1; }
    size_t dim2() const { return d2; }

    Slice_iter<double> row(size_t i);
    Cslice_iter<double> row(size_t i) const;

    Slice_iter<double> column(size_t i);
    Cslice_iter<double> column(size_t i) const;

    double& operator()(size_t x, size_t y);           // Fortran风格的下标
    double operator()(size_t x, size_t y) const;

    Slice_iter<double> operator()(size_t i) { return row(i); }
    Cslice_iter<double> operator()(size_t i) const { return row(i); }

    Slice_iter<double> operator[](size_t i) { return row(i); }    // C风格的下标
    Cslice_iter<double> operator[](size_t i) const { return row(i); }

    Matrix& operator*=(double);

    valarray<double>& array() { return *v; }
};
```

一个`Matrix`的表示就是一个`valarray`，我们通过切割将维数强加于这个数组。如果需要的话，我们也可以将有关表示看做是一维、二维、三维的等等，可以采用与这里通过`row()`和`column()`观察两个维同样的方式。这里利用`Slice_iter`绕过对`slice_array`的复制禁令。我无法返回一个`slice_array`：

```
slice_array<double> row(size_t i) { return (*v)(slice(i, d2, d1)); } // 错误
```

所以我返回的是一个包含指向`valarray`的指针和`slice`本身的迭代器，而不是一个`slice_array`。

我们还需要另一个类，为“常量所用切割的迭代器”`Cslice_iter`来表述在`const Matrix`切割与非`const Matrix`切割之间的差异：

```
inline Slice_iter<double> Matrix::row(size_t i)
{
    return Slice_iter<double>(v, slice(i, d2, d1));
}

inline Cslice_iter<double> Matrix::row(size_t i) const
{
    return Cslice_iter<double>(v, slice(i, d2, d1));
}

inline Slice_iter<double> Matrix::column(size_t i)
{

```

```

    return Slice_iter<double>(v, slice(i*d2, d2, 1));
}

inline Cslice_iter<double> Matrix::column(size_t i) const
{
    return Cslice_iter<double>(v, slice(i*d2, d2, 1));
}

```

*Cslice_iter*的定义与*Slice_iter*完全一样,但它返回的是对其切割的元素的*const*引用。

其他成员操作都极其简单:

```

Matrix::Matrix(size_t x, size_t y)
{
    // 检查x和y是明智的
    d1 = x;
    d2 = y;
    v = new valarray<double>(x*y);
}

double& Matrix::operator()(size_t x, size_t y)
{
    return row(x)[y];
}

double mul(Cslice_iter<double>& v1, const valarray<double>& v2)
{
    double res = 0;
    for (int i = 0; i < v1.size(); i++) res += v1[i] * v2[i];
    return res;
}

valarray<double> operator*(const Matrix& m, const valarray<double>& v)
{
    valarray<double> res(m.dim1());
    for (int i = 0; i < m.dim1(); i++) res[i] = mul(m.row(i), v);
    return res;
}

Matrix& Matrix::operator*=(double d)
{
    (*v) *= d;
    return *this;
}

```

我提供了 (i, j) 来表述 *Matrix* 下标, 因为 $()$ 是一个单独的运算符, 也因为在数值社会里大部分人更熟悉这种记法。行的概念用 $()$ (在C和C++ 社会里) 更熟悉的 $[i][j]$ 记法提供:

```

void f(Matrix& m)
{
    m(1, 2) = 5;           // Fortran风格的下标
    m.row(1)(2) = 6;
    m.row(1)[2] = 7;
    m[1](2) = 8;           // 不希望的混合风格 (但能工作)
    m[1][2] = 9;           // C++风格的下标
}

```

使用*slice_array*来表述下标, 假定了有一个很好的优化系统。

将这些推广到任意元素的 n 维矩阵, 并附带一组合理的操作, 这个工作留做练习 (22.9[7])。

或许你对二维向量的第一个想法是某种类似下面的东西：

```
class Matrix {
    valarray< valarray<double> > v;
public:
    // ...
};
```

这当然也能工作（22.9[10]）。不过，这样做很难在效率和兼容性上满足高性能计算的需要，而又不陷入到比`valarray`加上`slice`所表示的层次更低级也更传统的层次中。

22.4.7 临时量、复制和循环

如果你构筑起了一个向量或者矩阵类，为了满足对性能特别挑剔的用户需要，你很快就会发现自己不得不面对三个相互关联的问题：

- [1] 必须尽可能地减少临时量的数目。
- [2] 必须尽可能地减少矩阵的复制。
- [3] 必须尽可能地减少在组合操作中对于同样数据的多次循环。

标准库并没有直接面对这些问题，但是，我可以勾勒出一项技术，利用它可以产生出高度优化的实现。

考虑 $U = M * V + W$ ，其中的 U 、 V 、 W 都是向量， M 是矩阵。一种初级的实现方式是为 $M * V$ 和 $M * V + W$ 引进临时向量，并复制 $M * V$ 和 $M * V + W$ 的结果。一个聪明的实现是调用一个函数 `mul_add_and_assign(&U, &M, &V, &W)`，它不引进任何临时量，不复制任何向量，并按照最小可能的次数去触动矩阵里的每一个元素。

除了不多的几种表达式之外，很少需要去做这种层次的优化。所以，对效率问题的一个简单解决办法就是提供类似 `mul_add_and_assign()` 一类的函数，并让用户在它们确有价值地方去调用之。然而，完全有可能设计出一种 `Matrix`，使得对于形式正确的表达式，该优化能自动地进行。也就是说，我们可以将 $U = M * V + W$ 处理为一种带有4个运算对象的运算符。这方面的基本技术已经在讨论 `ostream` 操控符时（21.4.6.3节）时展示过。一般来说，这种技术可以用于使 n 个二元运算符的行为就像是一个 $(n + 1)$ 元运算符。处理 $U = M * V + W$ 时需要引进两个辅助类。由于能够实行某些特别有效的优化技术，在某些系统上，这种技术很可能导致令人印象深刻的加速（比如说30倍）。

我们首先定义一个 `Matrix` 和一个 `Vector` 乘的结果：

```
struct MVmul {
    const Matrix& m;
    const Vector& v;

    MVmul(const Matrix& mm, const Vector& vv) : m(mm), v(vv) {}

    operator Vector(); // 求值并返回结果
};

inline MVmul operator*(const Matrix& mm, const Vector& vv)
{
    return MVmul(mm, vv);
}
```

这一“乘法”什么都不做，只是保存好到它的各个运算对象的引用， $M * V$ 的求值被推迟了。由 $*$ 产生的对象与许多技术领域中所说的闭包密切相关。与此类似，我们也同样处理出现加

Vector的情况:

```
struct MVmulVadd {
    const Matrix& m;
    const Vector& v;
    const Vector& v2;

    MVmulVadd(const MVmul& mv, const Vector& vv) : m(mv.m), v(mv.v), v2(vv) { }

    operator Vector(); // 求值并返回结果
};

inline MVmulVadd operator+(const MVmul& mv, const Vector& vv)
{
    return MVmulVadd(mv, vv);
}
```

这也就推迟了对 $M * V + W$ 的求值。现在我们必须保证，在将它赋给一个 **Vector** 时，能用一个很好的算法对它求值:

```
class Vector {
    // ...
public:
    Vector(const MVmulVadd& m) // 用m的结果初始化
    {
        // 分配元素等
        mul_add_and_assign(this, &m.m, &m.v, &m.v2);
    }

    Vector& operator=(const MVmulVadd& m) // 将m的结果赋给 *this
    {
        mul_add_and_assign(this, &m.m, &m.v, &m.v2);
        return *this;
    }
    // ..
};
```

现在， $U = M * V + W$ 将自动展开为

```
U.operator=(MVmulVadd(MVmul(M, V), W))
```

它又由于在线解析而变成了我们所希望的简单调用

```
mul_add_and_assign(&U, &M, &V, &W)
```

很清楚，这一做法彻底去掉了复制和临时量。除此之外，我们还可以用某种优化的方式写出 `mul_add_and_assign()`。当然，即使我们只是用某种相当简单的非优化方式写这个函数，它所具有的形式也会为优化系统提供很多的机会。

我引进了 **Vector**（而没有用 **valarray**）就是因为需要定义赋值（而赋值又必须是成员函数；11.2.2节）。然而，**valarray** 是作为 **Vector** 所用表示的一个有力的候选者。

这种技术的重要性在于，大部分真正在时间上极端重要的向量和矩阵运算，都是用不多几种简单语法形式完成的。采用这种方式去优化涉及到六七个运算对象的表达式，通常不会有什么收获，采用常规的技术（11.6节）就足够了。

这种技术所基于的想法，就是通过编译时的分析和闭包对象包装，将子表达式的求值传递到一个代表组合计算的对象里。这种技术可能应用于各种各样的问题，只要它们具有共同的性质，其中需要将一些信息片段收集到一起，而后在一个函数里求值。我把这种为推迟求


```

        cout << " , ";
        for (int j = 0; j < s.size() [1]; j++) cout << gslice_index(s, 0, j) << " "; // 列
    }

```

这将打印出0 4, 0 1 2。

按照这种方式, 具有两个 (长度, 跨步) 对偶的`gslice`描述了一个二维数组, 有3个 (长度, 跨步) 对偶的`gslice`描述了一个三维数组, 依次类推。用`gslice`作为`valarray`的下标, 就产生出`gslice_array`, 其元素由`gslice`描述。例如,

```

void f(valarray<float>& v)
{
    gslice m(0, lengths, strides);
    v[m] = 0; // 将0赋值给v[0]、v[1]、v[2]、v[4]、v[5]、v[6]
}

```

`gslice_array`提供了与`slice_array`同样的一组成员。特别地, `gslice_array`也不能直接由用户构造或者复制 (22.4.6节)。实际上, 一个`gslice_array`也就是用`gslice`作为一个`valarray`的下标的结果 (22.4.2节)。

22.4.9 屏蔽

`mask_array`提供了另一种描述`valarray`的子集的方式, 并使其结果看起来就像是一个`valarray`。在`valarray`的环境中, 一个屏蔽 (mask) 也就是一个`valarray<bool>`。将一个屏蔽用做一个`valarray`的下标时, 值为`true`的位表明对应的`valarray`元素将作为结果的一部分考虑。这就使我们能在一个`valarray`的某个子集上操作, 即使不存在简单的模式 (例如`slice`) 来描述这个子集。看下面例子:

```

void f(valarray<double>& v)
{
    bool b[] = { true, false, false, true, false, true };
    valarray<bool> mask(b, 6); // 元素0、3和5
    valarray<double> vv = cos(v[mask]); // vv[0] == cos(v[0]), vv[1] == cos(v[3]),
                                        // vv[2] == cos(v[5])
}

```

一个`mask_array`提供了与`slice_array`同样的一组成员。特别是`mask_array`也不能直接由用户构造或者复制 (22.4.6节), 一个`mask_array`也是将一个`valarray<bool>` 用做`valarray`的下标的结果 (22.4.2节)。用做屏蔽的`valarray`的元素个数不能多于以它作为下标的那个`valarray`的元素个数。

22.4.10 间接数组——indirect_array

一个`indirect_array`提供了一种方式, 使我们可以任意地对`valarray`取子集, 也可以任意排列元素。例如,

```

void f(valarray<double>& v)
{
    size_t i[] = { 3, 2, 1, 0 }; // 前4个元素, 按逆序
    valarray<size_t> index(i, 4); // 元素3、2、1、0 (按此顺序)
    valarray<double> vv = log(v[index]); // vv[0] == log(v[3]), vv[1] == log(v[2]),
                                        // vv[2] == log(v[1]), vv[3] == log(v[0])
}

```


如果某个下标描述了两次，我们在同一个操作中就会两次引用到`valarray`的同一个元素。这正是`valarray`所不允许的别名的一种情况。所以，如果下标重复，`indirect_array`的行为是无定义的。

一个`indirect_array`提供了与`slice_array`同样的一组成员。特别是`indirect_array`也不能直接由用户构造或者复制（22.4.6节）。一个`indirect_array`也只能是将一个`valarray<size_t>`用做`valarray`的下标的结果（22.4.2节）。用做下标的`valarray`的元素个数不能多于以它作为下标的那个`valarray`的元素个数。

22.5 复数算术

标准库按照11.3节所描述的`complex`类的思路提供了一个`complex`模板。需要将库`complex`做成模板，是为了满足基于各种不同标量类型的复数的需要。这里特别为使用`float`、`double`和`long double`作为标量类型提供了专门化。

`complex`模板定义在名字空间`std`里，由 `<complex>` 给出：

```
template<class T> class std::complex {
    T re, im;
public:
    typedef T value_type;

    complex(const T& r = T(), const T& i = T()) : re(r), im(i) {}
    template<class X> complex(const complex<X>& a) : re(a.real()), im(a.imag()) {}

    T real() const { return re; }
    T imag() const { return im; }

    complex<T>& operator=(const T& z); // 赋值complex(z, 0)
    template<class X> complex<T>& operator=(const complex<X>&);
    // 类似地：+=, -=, *=, /=
};
```

这里的表示方式和在线函数都只是为了说明情况。人们也可以设想标准库的`complex`采用的是完全不同的表示方式。注意，这里用了一些成员模板，以保证能用任何`complex`类型对其他`complex`做初始化和赋值（13.6.2节）。

在本书里，我一直将`complex`作为一个类而不是一个模板。这也是可行的，因为我假设玩了一点名字空间花招，以取得我通常所喜爱的`double`的`complex`：

```
typedef std::complex<double> complex;
```

这里定义了常规的一元和二元运算符：

```
template<class T> complex<T> operator+(const complex<T>&, const complex<T>&);
template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);

// 类似地：-, *, /, == 和 !=

template<class T> complex<T> operator-(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);
```

提供了坐标函数：

```
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);

template<class T> complex<T> conj(const complex<T>&);
```

```
// 从极坐标 (abs(), arg()) 构造
template<class T> complex<T> polar(const T& rho, const T& theta);

template<class T> T abs(const complex<T>&); // 有时称为rho
template<class T> T arg(const complex<T>&); // 有时称为theta
template<class T> T norm(const complex<T>&); // abs() 的平方
```

提供了一组常用数学函数:

```
template<class T> complex<T> sin(const complex<T>&);
// 类似地: sinh, sqrt, tan, tanh, cos, cosh, exp, log 和 log10

template<class T> complex<T> pow(const complex<T>&, int);
template<class T> complex<T> pow(const complex<T>&, const T&);
template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
template<class T> complex<T> pow(const T&, const complex<T>&);
```

最后, 提供了流I/O:

```
template<class T, class Ch, class Tr>
basic_istream<Ch, Tr>& operator>>(basic_istream<Ch, Tr>&, complex<T>&);
template<class T, class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>&, const complex<T>&);
```

复数将以 (x, y) 的格式写出去, 可以按x、(x) 或者 (x, y) 的形式读入 (21.2.3节、21.3.5节)。提供了专门化`complex<float>`、`complex<double>` 和 `complex<long double>`, 以便限制转换 (13.6.2节), 并为优化实现提供可能性。例如,

```
template<> class complex<double> {
    double re, im;
public:
    typedef double value_type;

    complex(double r = 0.0, double i = 0.0) : re(r), im(i) {}
    complex(const complex<float>& a) : re(a.real()), im(a.imag()) {}
    explicit complex(const complex<long double>& a) : re(a.real()), im(a.imag()) {}

    // ...
};
```

现在, 一个`complex<float>` 就能隐式地转换为一个`complex<double>`, 而`complex<long double>` 则不行。类似的专门化保证`complex<float>` 和`complex<double>` 能隐式地转换为一个`complex<long double>`, 但是`complex<long double>` 却不能隐式地转换到`complex<float>` 或者`complex<double>`, `complex<double>` 也不能隐式地转换到`complex<float>`。但很奇怪的是, 这里没有为赋值提供与构造函数类似的保护。例如,

```
void f(complex<float> cf, complex<double> cd, complex<long double> cld, complex<int> ci)
{
    complex<double> c1 = cf; // 可以
    complex<double> c2 = cd; // 可以
    complex<double> c3 = cld; // 错误: 可能截断
    complex<double> c4(cld); // 可以: 显式转换
    complex<double> c5 = ci; // 错误: 不存在转换

    c1 = cld; // 可以, 但当心: 可能截断
    c1 = cf; // ok
    c1 = ci; // ok
}
```

22.6 通用数值算法

在 `<numeric>` 里, 标准库还按照 `<algorithm>` 中非数值算法的风格 (第18章), 提供了几个通用的数值算法:

通用数值算法 <code><numeric></code>	
<code>accumulate()</code>	积累在一个序列上运算的结果
<code>inner_product()</code>	积累在两个序列上运算的结果
<code>partial_sum()</code>	通过在序列上的运算产生序列
<code>adjacent_difference()</code>	通过在序列上的运算产生序列

这些算法推广了一些常用运算, 例如, 将求和推广到允许应用于任何种类的序列, 而且将应用于元素的运算作为参数。对于每种算法, 通用版本也都以能够应用于该算法最常见运算符的版本作为补充。

22.6.1 累积——`accumulate`

`accumulate()` 算法可以理解为向量的元素求和的推广。`accumulate()` 算法定义在名字空间 `std` 里, 由 `<numeric>` 给出:

```
template <class In, class T> T accumulate(In first, In last, T init)
{
    while (first != last) init = init + *first++; // 加
    return init;
}

template <class In, class T, class BinOp> T accumulate(In first, In last, T init, BinOp op)
{
    while (first != last) init = op(init, *first++); // 通用运算
    return init;
}
```

`accumulate()` 的简单版本采用它们的 `+` 运算符对序列中的所有元素求和。例如,

```
void f(vector<int>& price, list<float>& incr)
{
    int i = accumulate(price.begin(), price.end(), 0); // 累积int
    double d = 0;
    d = accumulate(incr.begin(), incr.end(), d); // 累积double
    // ...
}
```

请注意这里传递的初始值如何决定返回值的类型。

并不是我们想求和的所有东西都是某个序列里的元素。对于非序列元素的情况, 我们通常提供一个操作, 让 `accumulate()` 调用它去取得需要加的数据项。需要传递的这类操作中最明显的例子就是那种从数据结构提取值的操作。例如:

```
struct Record {
    // ...
    int unit_price;
    int number_of_units;
};

long price(long val, const Record& r)
{
    // ...
}
```

```

        return val + r.unit_price * r.number_of_units;
    }

    void f(const vector<Record>& v)
    {
        cout << "Total value: " << accumulate(v.begin(), v.end(), 0, price) << '\n';
    }

```

在某些社团里，与`accumulate`类似的操作被称做`reduce`或者`reduction`。

22.6.2 内积——`inner_product`

在一个序列里累积是很常见的，从一对序列中累积也不少见。`inner_product()`算法定义在名字空间`std`里，由`<numeric>`给出：

```

template <class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
{
    while (first != last) init = init + *first++ * *first2++;
    return init;
}

template <class In, class In2, class T, class BinOp, class BinOp2>
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
    while (first != last) init = op(init, op2(*first++, *first2++));
    return init;
}

```

如常，这里也只把第二个序列的起点作为参数传递，并假定这个序列至少与第一个序列一样长。

`Matrix`和`valarray`相乘的关键操作就是`inner_product`：

```

valarray<double> operator* (const Matrix& m, valarray<double>& v)
{
    valarray<double> res(m.dim1());
    for (int i=0; i<m.dim1(); i++) {
        Slice_iter<double>& ri = m.row(i);
        res[i] = inner_product(ri.begin(), ri.end(), &v[0], 0);
    }
    return res;
}

valarray<double> operator* (valarray<double>& v, const Matrix& m)
{
    valarray<double> res(m.dim2());
    for (int j=0; j<m.dim2(); j++) {
        Slice_iter<double>& cj = m.column(j);
        res[j] = inner_product(&v[0], &v[v.size()], cj.begin(), 0);
    }
    return res;
}

```

某些形式的`inner_product`被称为“点积”。

22.6.3 增量变化

`partial_sum()`和`adjacent_difference()`算法的功能互逆，用于处理增量变化的概念。它

们也定义在名字空间`std`里, 由 `<numeric>` 给出:

```
template <class In, class Out> Out adjacent_difference(In first, In last, Out res);
template <class In, class Out, class BinOp>
    Out adjacent_difference(In first, In last, Out res, BinOp op);
```

给了序列`a`, `b`, `c`, `d`等, `adjacent_difference()`将产生`a`, `b-a`, `c-b`, `d-c`等。

考虑一个温度读数的向量, 我们可以按如下方式将它转换为一个温度变化的向量:

```
vector<double> temps;
void f()
{
    adjacent_difference(temps.begin(), temps.end(), temps.begin());
}
```

例如, 17, 19, 20, 20, 17将转变为17, 2, 1, 0, -3。

与此相对应, `partial_sum()`使我们可以计算出一组增量变化的最终值:

```
template <class In, class Out, class BinOp>
Out partial_sum(In first, In last, Out res, BinOp op)
{
    if (first==last) return res;
    *res = *first;
    T val = *first;
    while (++first != last) {
        val = op(val, *first);
        *++res = val;
    }
    return ++res;
}
template <class In, class Out> Out partial_sum(In first, In last, Out res)
{
    return partial_sum(first, last, res, plus);    // 18.4.3节
}
```

给了序列`a`, `b`, `c`, `d`等, `partial_sum()`将产生`a`, `a+b`, `a+b+c`, `a+b+c+d`等。例如,

```
void f()
{
    partial_sum(temps.begin(), temps.end(), temps.begin());
}
```

请注意, `partial_sum()`的方式是先增加`res`, 而后才通过它赋新值。这就使`res`可以与该函数的输入是同一个序列。`adjacent_difference()`的行为方式与此类似。这样,

```
partial_sum(v.begin(), v.end(), v.begin());
```

将序列`a`, `b`, `c`, `d`转为`a`, `a+b`, `a+b+c`, `a+b+c+d`; 而

```
adjacent_difference(v.begin(), v.end(), v.begin());
```

能将它转回到最初的情况。作为特例, `partial_sum()`将把17, 2, 1, 0, -3转回17, 19, 20, 20, 17。

这些操作对于分析一系列的变化非常有用。例如, 分析股票值的变化, 所涉及的就是这两个操作。

22.7 随机数

随机数对于许多模拟和游戏都是必不可少的。在 `<cstdlib>` 和 `<stdlib.h>` 里，标准库为生成随机数提供了一个简单的基础：

```
#define RAND_MAX implementation_defined /* 大的正整数 */
int rand();                          // 在0与RAND_MAX间的伪随机数
void srand(unsigned int i);          // 将随机数生成器的种子置i
```

做出一个好的随机数生成器并不容易，不幸的是，并非所有系统都提供了好的 `rand()`。特别是随机数的低位常常很可疑，所以 `rand() % n` 并不是很好的产生 0 到 $n-1$ 之间随机数的可移植方式，采用 `int((double(rand()) / RAND_MAX) * n)` 常常能给出可接受的结果；然而应该认真地使用这个公式，我们必须当心结果为 n 的微小可能性。

对 `srand()` 的调用将从给定的参数种子开始给出一个新的随机数序列。为排除程序错误，一个重要因素就是从某个种子值开始的随机数序列总是重复的。然而，我们也常常希望用一个新种子开始每次新的运行。事实上，要使游戏不可预测，从程序的环境中取得一个种子常常很有意义。对于这类程序，实时时钟的某些位常可以作为很好的种子。

如果你必须写出自己的随机数生成器，那么一定要仔细测试它（22.9[14]）。

将随机数生成器表示成一个类将会更有用。按照这种方式，很容易构造出具有不同分布性质的随机数生成器：

```
class Randint { // 均匀分布，假定32位long
    unsigned long randx;
public:
    Randint(long s = 0) { randx=s; }
    void seed(long s) { randx=s; }

    // 魔幻数选用32位long中的31位
    long abs(long x) { return x&0x7fffffff; }
    static double max() { return 2147483648.0; } // 注意：double
    long draw() { return randx = randx*1103515245 + 12345; }

    double fdraw() { return abs(draw())/max(); } // 在区间 [0, 1]
    long operator()() { return abs(draw()); } // 在区间 [0, pow(2, 31))
};

class Urand : public Randint { // 均匀分布，区间 [0:n[
    long n;
public:
    Urand(long nn) { n = nn; }

    long operator()() { long r = n*fdraw(); return (r==n) ? n-1 : r; }
};

class Erand : public Randint { // 指数分布随机数生成器
    long mean;
public:
    Erand(long m) { mean=m; }
    long operator()() { return -mean * log( (max()-draw())/max() + .5); }
};
```

下面是一个简单测试：

```
int main()
{
    Urand draw(10);
    map<int, int> bucket;
    for (int i = 0; i < 1000000; i++) bucket[draw()]++;
    for (int j = 0; j < 10; j++) cout << bucket[j] << '\n';
}
```

除非每个**bucket**里的值都接近100 000，否则一定是程序的某处有错误。

这些是我所发布的最早的C++ 库（实际上是第一个“带类的C”库，1.4节）中随机数生成器经过略为修改后的版本。

22.8 忠告

- [1] 数值问题常常很微妙。如果你对数值问题的数学方面不是100%有把握，请去找专家或者做试验；22.1节。
- [2] 用**numeric_limits**去确定内部类型的性质；22.2节。
- [3] 为用户定义的标量类型描述**numeric_limits**；22.2节。
- [4] 如果运行时效率比对于操作和元素的灵活性更重要的话，那么请用**valarray**去做数值计算；22.4节。
- [5] 用切割表述在数组的一部分上的操作，而不是用循环；22.4.6节。
- [6] 利用组合器，通过清除临时量和更好的算法来获得效率；22.4.7节。
- [7] 用**std::complex**做复数算术；22.5节。
- [8] 你可以把使用**complex**类的老代码通过一个**typedef**转为用**std::complex**模板；22.5节。
- [9] 在写循环从一个表出发计算某个值之前，先考虑一下**accumulate()**、**inner_product()**、**partial_sum()**和**adjacent_difference()**；22.6节。
- [10] 最好是用具有特定分布的随机数类，少直接用**rand()**；22.7节。
- [11] 注意使你的随机数充分随机；22.7节。

22.9 练习

1. (*1.5) 写一个功能像22.4.3节中的**apply()**的函数，但让它不是成员并接受函数对象。
2. (*1.5) 写一个功能像22.4.3节中的**apply()**的函数，但让它不是成员、接受函数对象，而且修改其**valarray**参数。
3. (*2) 完成**Slice_iter**（22.4.5节）。在定义析构函数时应特别注意。
4. (*1.5) 用**accumulate()**重写17.4.1.3节的程序。
5. (*2) 为**valarray**实现I/O运算符<<和>>。实现一个**get_array()**函数，它由输入创建出一个**valarray**，大小的描述也作为输入中的一部分。
6. (*2.5) 定义和实现一个三维矩阵，带有适当的操作。
7. (*2.5) 定义和实现一个**n**维矩阵，带有适当的操作。
8. (*2.5) 实现一个类似**valarray**的类，并为它实现+和*。将它的性能与你所用的C++实现中的**valarray**做一个比较。提示：请在你的测试实例中包括 $x = 0.5(x+y) + z$ ，并用不同大小的向量**x**、**y**和**z**做试验。
9. (*3) 实现Fortran风格的数组**Fort_array**，其下标从1开始，而不是从0开始。

10. (*3) 实现 **Matrix**，它用一个 **valarray** 作为成员表示矩阵元素（而不是用一个到 **valarray** 的指针或者引用）。
11. (*2.5) 借助于组合器（22.4.7节），用 `[]` 实现一种高效的多维下标。例如，`v1[x]`、`v2[x][y]`、`v3[x][y][z]`、`v3[x][y]` 和 `v3[x]` 都应该通过一个简单的下标运算，产生出对应的元素或者子数组。
12. (*2) 将22.7节中的程序的思想推广到一个函数，给定一个随机数生成器作为参数，它能打印出其分布的一种简单图形表示，这可用做生成器正确性的粗略检查。
13. (*1) 如果 `n` 是 `int`，`(double(rand()) / RAND_MAX) * n` 的分布是什么？
14. (*2.5) 在一个正方形输出区域里打点。坐标对应该由 `Urand(N)` 产生，这里 `N` 是输出区域边上的像素个数。这个输出告诉你的由 `Urand` 产生的数的分布是什么？
15. (*2) 实现正态分布生成器 `Nrand`。

第四部分 用C++做设计

这一部分将在软件开发的更大背景中介绍C++ 及其支持的技术，其焦点在于设计，以及基于语言的各种结构去有效地实现这些设计。

章目

- 第23章 开发和设计
- 第24章 设计和编程
- 第25章 类的作用

“……我刚刚开始发现在纸上表述人的想法的困难性。只要想法是孤立的，描述它就非常容易；但在推理介入进来之处，为形成正确的联系、一些清晰性加上一种适度的流畅，就像我刚才所说的，对于我就成为说不清楚的困难了。”

—— 查尔斯·达尔文

第23章 开发和设计

没有银弹。[⊖]

——F. Brooks

构筑软件——目的与手段——开发过程——开发循环——设计目标——设计步骤——发现类——描述操作——描述依赖性——描述界面——类层次结构的重组——模型——试验和分析——测试——软件维护——效率——管理——重用——规模——个人的重要性——混成设计——参考文献——忠告

23.1 概述

本章是介绍软件生产的三章中的第一章。这个讨论将逐渐细化，从有关设计的相对高层次的观察开始，结束于直接支撑这些设计的C++ 特殊程序设计技术和概念。在引言和23.3节中关于目的与手段的简短讨论之后是本章的两个主要部分：

23.4节 有关软件开发过程的一种观点。

23.5节 有关软件开发过程组织的实践性评述。

第24章将讨论设计与程序设计语言之间的关系。第25章将从设计的观点出发，介绍类在软件组织中所扮演的一些角色。总而言之，第四部分中这三章的目标，就是填补所谓的独立于语言的设计和程序设计之间裂隙，这方面细节尚未引起足够的重视。这个连续谱的两端在大项目中都有其重要位置，但为了避免灾难和高昂的代价，它们必须成为一些具有连续性的概念和技术的有机组成部分。

23.2 引言

构造出任何非平凡的软件片段都是一项很复杂的而且常常使人气馁的工作。即使是对于作为个人的程序员，实际写程序语句也仅仅是这个过程的一部分。在典型情况下，分析问题、程序的整体设计、写文档、测试以及维护，还有对全部这些事务的管理，大量的工作使写出代码片段及排除其中错误的活动相形见绌。当然，有人也可以简单地将所有这些活动都贴上“程序设计”的标签，并随之做出一种富于逻辑性的断言，“我不做设计，我只是编程”。但是，无论人们如何称呼这些活动，关注其中各个独立的部分都是极其重要的——就像偶尔需要去关注这整个过程一样重要。在突击工作使系统得以发布的过程中，绝不应该丢掉了其中的细节或大的背景——然而这些正是经常发生的事情。

本章将集中关注程序开发过程中不涉及到写出代码片段及排除其中错误的那些部分。与本书其他地方有关个别语言特征和特定程序设计技术的讨论相比，这里的讨论不太精确，也

⊖ 意为：没有无坚不摧的利器，没有万能的灵药。——译者注

没有那么细致。这是必需的，因为并不存在创造出好软件的烹调手册。对于某些特定的已经深入理解的应用，有可能存在详细的“怎样做”的描述，而对更一般的应用则没有这种东西。任何东西都不可能代替程序设计中的智慧、经验和鉴赏力。由于这些，本章将只能提供一些一般性的忠告、可以选择的途径，以及警示性的观点。

这个讨论受到软件的抽象性质的阻碍，事实上，一些对小项目（比如说，一个或两个人写10 000行代码）行之有效的技术，却未必能扩展后用到中型或者大型项目中。由于这个原因，这里的一些讨论将采用与抽象的工程实践相类比的方式进行，而不是通过代码实例。请记住，“通过类比证明”是有意的欺骗，所以，这里只是想借助于类比阐述一些问题。采用C++ 特定术语，并附以实例的关于设计问题的讨论留待第24章和第25章里去做。本章所表述的思想既反映在C++ 语言本身的设计中，也反映在贯穿本书的各个具体实例的讨论里。

还要请读者记住在应用领域、人、程序开发环境等方面异乎寻常的多样性，你绝不要期望这里所提出的每个观点都能直接应用于你当前的问题。这些观点来自真实生活中的项目，能够应用于范围广泛多变的许多情况，但绝不应该认为它们是放之四海而皆准的。在考察这些观点时，请一定带着一种健康的怀疑态度。

C++ 可以简单地作为一种更好的C，但那样做，实际上就是将最有力的技术和语言特征弃之不顾，而只获取由使用C++中能够得到的潜在利益中很小的一部分。本章将集中关注那些能有效地利用C++ 的数据抽象和面向对象程序设计功能的设计方法，这种技术通常被称为面向对象的设计。

贯穿本章的若干主要论题是：

- 软件开发中最重要的一个方面就是弄清楚你试图去构造的东西。
- 成功的软件开发是一项长期活动。
- 我们开发的系统倾向于位于我们和我们所用的工具能够处理的复杂性的边缘上。
- 不存在能够代替设计和编程中的智慧、经验和鉴赏力的“烹调手册”方法。
- 试验对所有非平凡的软件开发都是必不可少的。
- 设计和编程是相互交叉、互相影响的活动。
- 软件项目的各个阶段，例如设计、编程和测试，不可能严格分离。
- 如果没有对编程和设计活动的管理，就无法考虑这些活动。

很容易（但常常是代价沉重的）低估这些观点的重要性。很难将它们蕴于其中的抽象思想转变成实际活动。这里特别要提出的是经验。就像造小船、骑自行车和编程一样，设计也不是一种只通过理论学习就能够掌握的技能。

一些情况太常见了，我们往往忘记了系统构筑过程中人的因素，而将软件开发简单地看做是“一系列具有良好定义的步骤，每一步都按照预先定义好的规则，对输入执行一些特定动作，产生所需要的输出”。这句话完全忽略了所涉及的人！设计和编程都是人的活动，忘记这一点就会丧失一切。

本章将特别关心这样一类系统的设计，相对于构造它们的人们的经验和资源而言，完成这些系统需要竭尽全力。这看起来正是个人或者组织试图去做那种位于他们能力极限上的项目时所遇到的情况。如果一个项目没有提出这种挑战，那么就不需要认真地讨论设计，这类项目早已有现成的框架，没有必要去颠覆它。只有在某种东西需要竭尽全力才可能完成时，人们才需要采纳新的更好的工具或者过程。在选定项目时也常常有这种倾向，那些相对的新

手常常认为“我们知道怎么做”，但实际上却不是这样。

绝没有设计和构筑所有系统的“惟一正确道路”。我确实希望把有关“惟一正确道路”的想法看做一种幼稚病，但许多有经验的程序员和设计师也经常趋附于这种认识。请记住，只因为某种技术对于你在去年可行，或者对于一个项目可行，并不意味着它可以不加修改地适用于其他人或不同的项目。保持一种开放的心态是最重要的。

很清楚，这里的大部分讨论将与大规模软件的开发有关。未涉足这种开发的读者可以舒舒服服地坐下来，愉快地观看自己悠然处于事外的那些恐怖场面。换种方式，他们也可以只去看这个讨论中与个人工作方式有关的那些部分。关于什么时候就必须先做设计而后再编程，在代码规模上并不存在一个下限。不过，用于设计和文档的每种途径都有其适用下限，请看23.5.2节有关规模的讨论。

软件开发中最基本的问题就是复杂性。只存在一种对付复杂性的基本方法：分而治之。如果一个问题可以分成两个能分别处理的子问题，这个划分就已经解决了问题的一多半。这一简单原理可能以令人吃惊的丰富多彩的方式去应用。特别是，在系统设计中使用一个模块或者类，就把程序分成了两部分（实现与其用户），它们之间只通过一个（理想情况）定义良好的界面相互联系。这是处理程序中内在复杂性的基本方式。与此类似，设计一个程序的过程也可以分割成一些独立活动，在所涉及的人之间（理想情况）也有定义良好的相互关系。这就是处理开发过程的和所涉及人员的内在复杂性的基本途径。

在两种情况下，各个部分的选择以及不同部分之间界面的刻画都是最需要经验和鉴赏力的地方。这种选择不是一种简单的机械性过程，通常都需要某种洞察力，而这种洞察力的获得，只能通过在某个适当抽象层次上对一个系统的透彻理解（参见23.4.2节、24.3.1节、25.3节）。对程序或者软件开发过程的短视观点经常导致带有严重缺陷的系统。还应当注意，无论是对人还是对程序，分开总是容易的，更困难的部分是保证分界两边各个部分间的有效通信，而又不破坏这种分解，不抑制为协作所需要的通信。

本章将介绍的是一种设计途径，而不是一种完整的设计方法。某种完全形式化的设计方法已超出了本书的范围。这里介绍的途径可能被用于不同程度的形式化，或者作为不同形式化的基础。同样，本章不是一个文献综述，不想去触及与软件开发有关的每一个论点或者介绍每一种观点。这同样也超出了本书的范围。有关文献的综述可以在 [Booch,1994] 里找到。还请注意，这里对术语的使用也采取了最普通最方便的方式。最“有意思”的术语（例如设计、原型和程序员）在各种文献中都有许多不同的甚至相互冲突的定义。请小心，不要基于某些术语的特定的或者具有局部精确性的定义，从这里所写的文字中读出了某些并不是这里原意的东西。

23.3 目的与手段

专业程序设计的目标就是发布满足用户需要的产品。达到这一目的的基本手段就是生产出具有清晰内部结构的软件，培养出一批设计师和程序员，并使之具有足够的技能和动力去快速有效地应对各种变化和机遇。

为什么？程序的内部结构和将它开发出来的过程，按照理想情况，都是最终用户不必关心的。说得更重一点，如果最终用户不得不关注程序是怎样写出来的，那么这个程序一定有什么地方出了毛病。承认了这些事实之后，一个程序的结构以及创建它的人员的重要性又体

现在哪里呢？

程序需要有一个清晰的结构，以便较容易：

- 调试。
- 移植。
- 维护。
- 扩展。
- 重组。
- 理解。

这里的重点是，软件中每个成功的主要部分都会有一个很长的生命期，一系列程序员和设计师将在它上面工作，移植到新的硬件，去适应原本没有预料到的用户，并反复地重新组织。贯穿着这个软件的整个生命期，它的新版本必须被生产出来，带着可接受的错误率，而且要及时。没有为这一点做计划就是会失败。

注意，尽管按照理想的情况，最终用户不必知道一个系统的内部结构，但他们实际上也可能希望知道。例如，一个用户可能希望知道某系统的设计细节，以便能评价它大致的可靠性以及修改和扩充潜力。如果所论及的软件不是一个完整系统，而是为构造其他软件的一组库，那么用户将希望知道更多的“细节”，以便能更好地使用这些库，并更好地将它们作为获取思想的资源。

必须努力在缺乏对软件整体设计和过度强调结构之间寻找一种平衡。前者将引起无穷的修改（“我们将立即发布这一个，并将在下一个版本里解决这个问题”）。后者导致一个过于精致的设计，使最具根本性的东西被淹没在形式主义之中，还由于不停地重新组织而导致实现的延误（“但这种新结构比原来的那个好得多，人们会愿意等着它的”）。这经常会导致系统对资源的需求非常强烈，以至于大部分潜在用户都无法承受其代价。这种平衡总是在设计中最困难的方面出现，这也正是才能和经验发挥作用的地方。对作为个人的设计师和程序员，做出这种选择也是很困难的；对于涉及到许多具有不同技能水平的人员的大型项目，做出这种选择就更加困难。

一个程序需要由一个组织生产出来并继续维护，它应该能完成这件事情，无论其人事、方向和管理结构发生了什么变化。对付这一问题有一种很流行的方式，那就是试图将系统的开发归约到能够塞进一个严格框架的一些相对低层次的任务里。也就是说，其基本思想是创建一支容易训练的（廉价的）并具有互换性的低水平程序员（“编码员”）队伍和另一个不那么廉价但同样具有互换性（因此有同样可依赖性）的设计师队伍。假定编码员不做任何设计决策，而且假定设计师本人都不涉足卑微的代码细节。这种途径经常失败。在它还能行的地方，将生产出性能较差且过于庞大的系统。

这种途径的问题在于：

- 实现者与设计者之间不充分的通信。这将导致机会的丧失、延期、低效率，以及缺乏从经验中学习而引起的种种问题。
- 未给实现者的主动精神提供充分的发展空间。这将导致专业培养不足、缺乏原动力、懒散和很高的人员流动比率。

简而言之，这样一种系统缺乏反馈功能，不能使人们从其他人的经验中获益。它是对宝贵的人类才能的浪费。我们需要创建一种框架，使人们能在其中应用各自的聪明才智，开发出新的技

能, 贡献各种思想, 并为自己所完成的东西不仅能说得过去, 而且很实际很经济而感到愉快。

另一方面, 如果没有某种形式结构, 一个系统也不可能无限期地构筑、写文档、维护下去。简单地找到一批最好的人, 让他们按自己所想的最佳方式去专攻某个问题, 对于一个需要创新性的项目, 这常常是一个很好的起点。然而, 随着项目的进展, 在项目中引进更多安排好的、专门的、更形式化的交流就变得越来越必要了。对于“形式化”, 我的意思并不是某些数学的或者能够机械验证的记法(虽然这很好, 在那些可用的且可以应用的地方), 而是一组有关记法、命名、文档书写、调试等的指导性规则。再说一次, 取得平衡和有意识的适度很有必要。过分严格的系统将阻碍成长, 窒息创新。在这种情况下, 被测试的将是管理者的才能和经验。对于个人而言, 与之等价的两难问题是要做出选择: 在哪里试着做得更聪明些, 而在哪里就简单地“按书本行事”。

这里的基本建议是, 我们不只要为当前项目的下一个版本做计划, 还要考虑更长远的问题。只计划下一个版本就是计划失败。我们必须面向生产和管理许多项目的许多版本去发展组织结构和软件开发策略, 我们必须计划一系列的成功。

“设计”的宗旨在于为程序创建起一种清晰且相对简单的内部结构, 有时被称为体系结构。换句话说, 我们希望建立起一个框架, 使单独的代码片段可以纳入其中, 并能对这些单独代码片段的书写起指导作用。

设计是有关的设计过程的最终产品(就算是一个不断重复的过程, 也存在着一个最终产品)。它是设计师和程序员之间、程序员之间交流的焦点。在这里有权衡轻重的能力就非常重要了。如果我作为一个个人程序员设计了一个小程序, 准备明天去实现, 其精确性和细节程度可能就只是在一个信封背面的随意几笔。而在另一个极端, 在开发一个涉及到数以百计的设计师和程序员的系统时, 就可能需要用形式化或者半形式化的记法, 仔细写出很多本规范。为设计确定一种恰当层次的细节、精确程度和形式方式, 这本身也是一项富于挑战性的技术和管理任务。

在本章和随后的两章里, 我假定一个系统的设计被表述为一组类声明(典型情况是将它们的私用声明忽略, 作为假想的细节)及其相互关系。这是一种简化。在一个特定设计中还要加入比这多得多的细节, 例如, 并发性、名字空间管理、非成员函数和数据的使用、参数化类和函数、为最少的重新编译面做的代码组织、持续性功能, 以及多种计算机的使用等。然而, 为了能在这个层次上讨论就必须做一些简化, 而类正是在C++ 里进行设计的最恰当的焦点。在这一章的讨论进程中将提到其中的一些问题, 那些直接影响C++ 程序的设计的问题将在第24章和第25章讨论。有关一种特殊的面向对象设计方法的细节讨论和实例, 可以参见[Booch, 1994]。

我没有去划清分析和设计之间的界线, 是因为对这个问题的讨论已经超出了本书的范围, 而且它对特定设计方法的变化也比较敏感。找到一种能够与设计方法相匹配的分析方法, 找到一种能够与程序设计风格和所用语言相匹配的设计方法都是最重要的事情。

23.4 开发过程

软件开发是一个不断重复的递进的过程。在开发中将不断地重新经历这一过程中的各个阶段, 每次经历时都通过精化产生出该阶段的最终产品。一般说, 这一过程既没有开始, 也没有结束。在设计和实现一个系统时, 你在其他人的设计、库和应用软件的基础上开始工作。

当你结束时，你留下了一个包括设计和代码的实体，供其他人去精化、修改、扩展和移植。自然，一个特定项目必然有其确定的开始和结束，将一个项目的时间和范围划分得清晰精确是极其重要的（但又常常困难得令人不可思议）。然而，自以为你是从一个清晰的状态开始却可能引起严重的问题，自以为那个世界终结在“最终交付”，将给你的继任者造成类似的严重问题，而这个继任者常常就是扮演着另一个角色的你自己。

这种情形蕴含着的一件事，就是下面几节可以按任意顺序阅读，因为在实际项目中，设计和实现的各个方面几乎能任意地交叉出现。也就是说，“设计”几乎总是在原来设计和某些实现经验基础上的重新设计。进一步说，这个设计又受到计划安排、所涉及的人员的能力、兼容性等因素的束缚。对于设计师/管理人员/程序员的一项主要挑战，就是为此过程建立一种秩序，而又不会窒息创新和破坏反馈循环，因为它们对于成功的开发都是必不可少的。

开发过程包含三个阶段：

- 分析：定义需要解决的问题的范围。
- 设计：建立系统的整体结构。
- 实现：写出代码并完成测试。

请切记这个过程不断重复的本性——这些阶段不是编号的，这一点非常重要。注意，程序开发还有一些重要方面没有作为单独的阶段出现，是因为它们应该弥漫在整个过程之中：

- 试验。
- 测试。
- 对设计和实现的分析。
- 撰写文档。
- 管理。

软件“维护”不过是穿过这一开发过程的更多循环罢了（23.4.6节）。

最重要的是，分析、设计和实现不应该过分地相互分离，应该使涉足其中的所有人员能共享一种文化，使他们能有效地交流。在大型项目中，实际情况却常常不是这样。理想的情况是，参与项目的个人应该从一个阶段转移到另一个阶段，因为传递精微信息的最好方式就是通过人的头脑。不幸的是，组织机构常常建立起某些阻碍，阻止这种转移。例如，给予设计师比“仅为程序员者”更高的地位或者更高的待遇。如果让人们移来移去参与学习和讲授的想法不能实现，那么至少也应鼓励他们与涉足同一开发中的“其他”步骤的人们开展常规性的交流讨论。

对于小型到中型的项目，在分析和设计之间常常没有清晰的划分，这两个阶段常常被合二为一。与此类似，在小项目中，设计和实现之间常常也没有分开。当然，这样就解决了交流的问题。重要的是，对给定项目应当有适度的形式规定，并维持各个阶段间适度的分离（23.5.2节）。并不存在做这些事情的惟一正确方法。

这里所描述的软件开发模型与传统的“瀑布模型”截然不同。在瀑布模型中，开发过程具有一种顺序的线性形式，开发从分析阶段一直到测试阶段。瀑布模型受到这种单向信息流动所造成的基本缺陷的损害。在“下游”发现问题时，通常存在着强有力的方法学上的和组织上的压力，要求去做局部修正，也就是说，强烈要求解决问题时不影响开发过程的前面阶段。缺乏反馈将导致有缺陷的设计，局部修正将导致扭曲的实现。当出现不可避免的情况，使信息必须流向源头并引起设计的改变时，结果只是一股极慢的拖累沉重的涓涓细流穿过整

个系统。该系统特别适合阻止这种改变，至多是极不情愿地极其缓慢地做出回应。这样，有关“不要改变”或者“局部修正”的论点就变成了另一种论点：一个子组织不能“为了它自己的方便”将大量的工作强加给其他的子组织。特别是，在发现了一个主要缺陷时，常常出现大量与决策缺陷有关的纸面工作，涉及修改文档的工作常大大超过修正代码所需要的工作。按照这种方式，纸面工作很可能变成软件开发中的主要问题。很自然，这一问题可能而且确实出现了，无论人们如何去组织大型系统的开发。话说回来，有些纸面工作毕竟是不可或缺的，但是，采用线性开发模型（一道瀑布）将极大地增加这一问题变得失去控制的可能性。

瀑布模型的问题在于不充分的反馈，以及无能力对修改做出响应；而这里勾勒出轮廓的交互式途径中也有危险。在这里存在着一种诱惑，使我们可能用一系列不收敛的变化去代替真正的思想和进步。这两个问题都很容易诊断，但却都很难解决。无论人们如何去组织一项工作，都很容易受到诱惑，将错误的行为当做进步。很自然，随着项目进展，对开发过程不同阶段的重视程度也应有所变化。开始时，重点是分析和设计，较少关心编程问题。随着时间的推移，更多的资源将移到设计和编程，而后将更多地关注编程和测试。但无论如何，关键点是绝不能在关注分析/设计/实现这个连续谱中某部分时，完全忽略了其他各个部分。

请记住，如果你对试图达到什么目标没有一个很清晰的想法，无论你将多少注意力放在细节上，使用的是多么合适的管理技术，采用了多少高级技术，都不会有多少帮助。由于缺乏定义清晰的实际目标而失败的项目比因为其他原因而失败的项目更多。无论你要做什么，也不管你怎样去做，请定义好一些实实在在的目标和阶段性标志，而且不要企图去为社会问题寻找技术性解决方案。另一方面，应当使用任何可用的合适技术——即使它涉及到投资。有了合适的工具，在合理的环境中，人们可以工作得更好。请不要欺骗自己，以为按照这些建议去做很容易。

23.4.1 开发循环

开发一个系统是一件不断重复的活动。其主要循环是通过如下序列的一再重复的旅行：

[0] 考察问题。

[1] 创建一个整体设计。

[2] 找出一些标准组件。

— 为本设计而定制这些部件。

[3] 创建一些新的标准组件。

— 为本设计而定制这些组件。

[4] 装配起整个设计。

作为类比，让我们来考虑一个汽车工厂。在一个项目开始时，需要有对一种新型汽车的总体设计，最先的决策将基于某些分析，用一些一般性的术语刻画这种汽车，主要是根据对该车所设想的应用，而并不牵涉如何才能达到所需要的性质的具体细节。确定需要有哪些性质（或者采取更好的方式，为确定究竟需要什么性质提供一个相对简单的准则）常常是一个项目中最困难的部分。如果这件事真能做好，那么通常总是由某个富于洞察力的个人独立完成的，这常常被称为一个透视（vision）。一个项目缺乏这种清晰目标的情况也是相当常见的，许多项目因此而踉踉跄跄以至于失败。

比如说，我们希望制造一种四门中型车，带有相当强劲的引擎。设计中的第一步通常都

不是从空白开始去设计这辆车（及其所有的零部件）。软件设计师和程序员也处于类似环境中，但却可能很不明智地想从空白开始。

第一步是考虑从工厂自身的库存和可靠供应商那里能够得到哪些组件，这样找到的组件并不一定正好适合这种新车。存在许多方式去改制这些组件，或许可能去影响这些组件的“下一个版本”的规范，使之更适合我们的项目。例如，可能存在一种具有合适特性的引擎，但它的输出功率却略显不足。我们或者引擎的提供商可以添加一个涡轮增压器，又不影响其基本设计。注意，如果引擎的原始设计并没有结合进某种定制的可能性，那么要做这种修改而又“不影响其基本设计”就未必可行了。这种改制通常需要你与你的引擎供应商之间的合作。软件设计师和程序员也有类似的选择机会。特别地，多态性的类和模板常常可用于有效地改制。当然，如果没有一点前瞻性，如果不能得到这种类的提供者的合作，我们就没指望去做任意的扩充。

在穷尽了合适的标准组件后，汽车设计师还不急于去为新车设计最优化的新组件，因为这样做代价会太高昂。假定不存在合适的空调装置，而在引擎舱里有一块合适的L形空间可以利用。一种解决方案就是设计一种L形的空调单元。但是，将这种独特设备用于其他车型的可能性将非常小——即使是经过高成本的定制。这也意味着，我们的汽车设计师将不能与其他汽车设计师共同承担这种单元的生产成本，而且这种单元的可用生命期也会比较短。根据这种认识，看来值得去设计能满足更广泛需求的单元，也就是说，设计一种在设计上比我们所假想的L型怪物更清晰也更适宜改制的空调单元。这样做，所涉及的工作通常会比做L型单元更多一些，甚至可能涉及到修改我们汽车的整体设计，以适应这种具有更一般用途的单元。又因为这种新单元的设计要比我们的L型奇异物件更加广泛可用，它也将需要做一点定制，才能正好符合我们修正之后的需要。同样，软件设计师和程序员也有类似的选择机会。也就是说，设计师可以不去写项目专用的代码，而是设计一种具有普遍意义的组件，这将使它成为某些世界中一个标准的候选品。

最后，当弄完了所有潜在的标准组件之后，我们去装配起“最后的”设计。我们尽可能少地采用了若干特殊组件，因为下一年我们将不得不为下一个新车型再次经历这个工作的某种变形，而这些特别设计的组件将是我们最可能重新去做或者丢掉的。非常糟糕，在传统软件设计中的经验是：一个系统里很少有几个部分能看做是可分离的组件，这些之中更少有什么可以用到它们原来的项目之外。

我并没有说所有汽车设计师都像我所勾勒的这一类比中那样富于理性，也没有说所有软件设计师都犯这里所提及的错误；相反，我想说的是，这个模型对软件也可行。特别是，本章及随后几章要介绍一些技术，可以使之对于C++可行。不过我也确实声明，由于软件的无形的本质，要避免这些错误是更困难的（24.3.1节、24.3.4节）。在22.5.3节里，我还要争辩说，企业文化常常妨碍人们去使用这里所描绘的模型。

注意，只有在你有长远打算之时，这里的开发模型才能真正工作得很好。如果你的视野只延伸到下一个版本，那么创建和维护标准组件就完全没有意义了，它只能被看做是一种臆造出来的额外负担。这个模型只是提供给那样一种组织，其生命期将跨过几个项目，其规模使得在工具方面（为了设计、编程和项目管理）的额外投资具有实际价值。这是一类软件工厂的蓝图。有意思的是，这种模型与最好的个人程序员的实践只有规模上的差异。这些人在多年中为提高个人的工作效率，创造起一大批技术、设计、工具和库。看起来，实际中大部

分组织机构都未能利用最好的个人实践活动的经验，因为缺乏远见，也因为没有能力在超过很小的规模之上管理这种实践性活动。

注意，期望“标准组件”成为全球性的标准并不合理。确实会出现少量国际性的标准库，但是，绝大部分将只能成为一个国家、一个产业、一个公司、一条生产线、一个部门、一个领域等内部的标准。这个世界太大，使我们不可能对所有组件和工具都实现全球性的标准，这也确实不应该成为一个目标。

在初始设计时就全球性作为目标，实际上就是命令这个项目永远也不能完成。开发循环之所以是一个循环，也因为取得一个能工作的系统是必不可少的，因为只有从那里才能取得经验（23.4.3.6节）。

23.4.2 设计目标

设计的总体目标是什么？当然，简单性是其中之一，但简单性又要依据什么准则呢？我们应假定一个设计将要演化，也就是说，这个系统将要扩展、移植、调整，一般说，将以某些不可能都事先预见到的方式改变。因此，我们就必须确定目标，使设计和实现出来的系统比较简单，有关的约束条件使它能够以多种方式变化。事实上，应该假定在它的初始设计到第一个版本间，这个系统就已改变了几次。这种假定是很现实的。

这也就意味着，系统的设计必须能在一系列变化之后仍然尽可能简单。我们必须为变化而设计，也就是说，我们必须将目标定在：

- 灵活性。
- 可扩充性。
- 可移植性。

解决这种问题的最好方式是将系统中可能变化的区域封装起来，并为设计师/程序员提供某些非侵入式的方法，使他们能够修改代码的行为。完成这项工作需要标识出应用中的关键性概念，给每个类一种排他性的责任，要求它维护着与某个单一概念有关的全部信息。在这种情况下，一项变化的实施就只需要修改与之相关的类。最理想的情况是，一个概念的修改可能通过一个派生类（23.4.3.5节），或者通过给某模板传递一个不同参数而完成。当然，陈述这种理想比实现它要容易得多。

现在考虑一个例子。在一个涉及天气现象的模拟中，我们希望显示出一种雨云。但怎样做呢？我们不能有一种通用例程来完成云的显示，因为云的视觉形象依赖于云的内部状态，而这种状态完全在云的职责范围之内。

对于这种问题的第一种解决方案就是让云去显示自己。这种风格的解法在许多受限的环境中都是可以接受的。但是它不够一般，因为存在许多观察云的方式：例如，作为一个细致的图形、作为一个粗略的轮廓或者作为地图上的一个图标。换句话说，云的视觉形象依赖于云和它的环境。

这一问题的第二种解决方案是让云意识到它的环境，而后让云去显示自己。在更多的环境里可以接受这种解决方案。但这仍然不是一种一般性的解法。让云了解其环境的这些细节也违反了前面的决断，即让一个类对一件事情负责，且每件“事情”都是某一个类的责任。很可能无法创造出一个具有内在和谐性的“云的环境”的概念，因为一般来说，云的视觉形象依赖于云和观察者。甚至在现实生活中，云的视觉形象也在很大程度上依赖于我怎么去看

它。例如，是通过我的裸眼、通过偏振滤镜，还是通过气象雷达。除了云雨观察者之外，某些“一般性的背景”，例如太阳的相对位置等，也必须予以考虑。此外，其他实体，例如其他的云和飞机等，将进一步使问题复杂化。如果再加上可以同时存在多个观察者，设计者的日子就更难过了。

第三种解决方案是让云以及其他物体（如飞机和太阳）将自己描述给环境。这一解法对于大部分目的而言都具有足够的一般性^①。然而，这样做可能带来复杂性和运行时间方面的显著代价。例如，我们将如何安排，使观察方可以理解由云和其他实体产生出的描述呢？

雨云在程序里并不经常出现（要找一个实际例子，请见15.2节），但涉及到许多I/O操作的对象则在程序里比比皆是。这就使这个有关云的例子与程序有了一般性的联系，特别是与库设计的联系。针对逻辑上类似的C++ 代码实例，可以在流I/O系统中格式化输出所用的操控符那里看到（21.4.6节、21.4.6.3节）。注意，这里的第三个解决方案并不就是“那个正确的解法”，而只是最具一般性的解法。设计师需要权衡在一个系统里的各种需要，去选择适合特定系统里针对特定问题的一般性和抽象层次。作为一种经验法则，一个长寿命程序的合适抽象层次就是你可以理解和负担的层次中最一般的，当然不是绝对的最一般。超出特定项目范围和参加者经验的一般性可能很有害，也就是说，它可能造成延期、无法接受的低效率、无法管理的设计，或者就是失败。

为使这种技术成为可管理的、经济的，我们必须为重用而设计和管理（23.5.1节），而且不能完全忽视效率（23.4.7节）。

23.4.3 设计步骤

考虑如何设计一个单独的类。通常这并不是一个好想法。概念并非孤立存在的，相反，一个概念通常总是定义在其他概念形成的环境之中。与此相似，一个类也不应孤立地存在，而应该与逻辑上相关的类一起定义。典型情况是人需要做出一个相关的类的集合，这样一个集合常被称为一个类库或一个组件。有时一个组件里的所有类组成了一个类层次结构，有时它们是一个名字空间的成员，有时它们不过是实际汇集在一起的一些声明（24.4节）。

在一个组件里的一组类总是通过某种逻辑准则结为一体的，常常是由于一种共同风格，常常由于依赖于一种公共的服务。这样，一个组件就是一种设计、写文档、拥有和重用的单位。这并不意味着如果你使用了某个组件里的一个类，你就必须理解和使用这个组件里所有的类，也不意味着可能要把该组件中每个类的代码都装入你的程序。正相反，我们总是要努力去保证，一个类的使用只给机器资源和人们的付出带来最小的开销。当然，为了使用一个组件里的任何一部分，我们将需要理解定义这一组件的逻辑准则（希望文档里写得足够清楚）、使用约定、该组件设计中所蕴涵的风格以及它的文档，还有那些公共服务（如果存在的话）。

现在考虑应该如何完成一个组件的设计。因为这常常是一项挑战性的工作，我们值得将它分解为一些步骤，以便能以一种逻辑的和完全的方式，集中关注其中的各项子工作。如常，做这件事也不是只有一条正确道路，不管怎样，下面是一些人所采用的一系列步骤：

[1] 找出概念/类及其最基本的相互关系。

^① 即使这一模型对于某些极端情况也可能不够，例如要求基于光线跟踪的高质量图形。我设想，要达到那样的细节，将要求设计师转移到另一个抽象层次去。

[2] 精化这些类，描述它们的一组操作。

- 将这些操作归类，特别是考虑所需要的构造函数、复制和析构函数。
- 考虑最小化、完全性和方便性。

[3] 精化这些类，描述它们之间的依赖关系。

- 考虑参数化、继承和使用方面的依赖关系。

[4] 描述界面。

- 将函数分为公用操作和保护操作。
- 描述这些类中各操作的确切类型。

注意，这也是在一个重复性过程中的一些步骤，要为初始实现或者重新实现产生一个用起来很舒服的设计，通常需要经过这一工作序列的几次循环。像这里所描述的很好完成的分析和数据抽象有一个优点，它能使重新安排类之间的关系变得相对容易些，即使是在代码已经写成之后。当然，这绝不会是一件平凡工作。

23.4.3.1 步骤1：发现类

找出概念/类及其最基本的相互关系。一项好设计的关键是直接模拟“真实世界”的某个侧面，也就是说，将应用领域中的概念拿来作为类，用良好定义的方式表示类之间的关系（例如用继承），并且在不同的层次上反复地这样做。但是我们怎么能找出这些概念？什么是确定我们所需要的类的实际途径呢？

开始寻找的最佳地方应该是在应用本身，而不是去看计算机科学家口袋里的抽象和概念。听听某些在系统完成后将成为专家用户的人，以及对现有的将被取代的系统不满意的人说些什么。注意他们所用的词汇。

人们常说名词将对应于程序里所需的类和对象，确实经常如此。但是，这并不意味着故事的结束。动词有可能指称对象上的操作、传统的基于其参数值产生新值的（全局）函数，或者甚至是类。作为最后一种情况的例子，请注意函数对象（18.4节）和操控符（21.4.6节）。像“重复”或者“提交”一类的动词可能用一个迭代器对象，或者代表数据库提交操作的对象表示。再考虑形容词“可存储的”、“并行的”、“注册的”和“约束的”，这些也可能成为一些类，以使设计师或程序员能通过描述虚基类，提供随后设计的类所需检取或者选择的属性（15.2.4节）。

并不是所有的类都与应用层中的概念相对应。例如，那些表示系统资源和实现层的抽象（24.3.1节）。避免过近地模拟老系统也非常重要。例如，我们不会希望一个以数据库为中心的系统亦步亦趋地重复一个手工系统的各个方面，原来系统的存在可能不过是为使人们能物理地将纸片移来移去。

继承用于表述概念之间的共性。最重要的是，用它表示一种基于各个类所代表的个别概念的层次性组织（1.7节、12.2.6节、24.3.2节）。这有时被称为分类（classification）或分类学（taxonomy）。共性必须去主动寻找，推广和聚类都是高层次的活动，需要借助于洞察力，才能取得有用且耐久的结果。公共基类应当代表某种一般性的概念，而不是某个与之相近的概念，只不过其中需要表示的数据恰好比较少。

请注意，这种分类应该基于所模拟系统中的概念的某些方面，而不是基于在别的领域中可能合法的一些东西。例如，在数学里圆是椭圆中的一类，但是在大部分程序里，圆不应该由椭圆派生，椭圆也不应由圆派生。常听到这样的论点，如“因为这就是数学里的情况”或

者“因为圆的表示是椭圆表示的子集”。这些都不是决定性的，常常都是错误的。这是因为，对于大部分程序而言，一个圆的关键属性是它有一个中心和到其圆周的固定距离，圆的所有行为（所有操作）必须维护这些属性（不变式，24.3.7.1节）。而在另一方面，椭圆则由两个焦点描述，在许多程序里这两个焦点可以独立地改变。如果焦点重合，这个椭圆看起来就像圆，但是它不是圆，因为它的操作并不能维持圆的不变式。许多系统里都对这种差异有所反应，方式是为圆和椭圆提供的操作集合相互都不为子集。

我们不会一下就想出了一组类和这些类间的相互关系，并将它们用于最终的系统。相反，我们会先创建出一组初始的类和关系，而后又反复对它们做精化（23.4.3.5节），以达到一组类关系，使之足够一般、灵活和稳定，确实能对系统的进一步演化有所帮助。

找出初始的关键性概念/类的最好工具就是黑板，对它们的初始精化的最好方法就是与应用领域的专家和一些朋友讨论。要开发出一组有活力的词汇和概念框架，这种讨论是必不可少的，少数几个人就可以独立地去做。为从初始的候选类集合中发展出一组有价值的类，一种方式就是去模拟一个系统，让设计师去扮演类的角色。这样做，能使初始想法中不可避免的荒谬情况暴露到一种开放的、使人兴奋的有关替代方案的讨论中，使人们在如何改变设计上取得共识。这种活动也可以用索引卡片支持，在卡片上记下文档。这种卡片通常被称为CRC卡片（类、职责和协作，“Class, Responsibility, and Collaboration”）。

一个用例（use case）就是关于一个系统的一次特定使用的描述。这里是某电话系统的用例的一个简单例子：将电话拿起，拨号，在另一端的电话振铃，另一端的电话被拿起。做出一组这样的用例对子开发的各个阶段都有巨大的价值。在开始阶段，找出这种用例能帮助我们认识准备去构造的是什么。在设计阶段，可以利用它们跟踪贯穿这个系统的路径（例如，使用CDC卡片），从某种用户观点检查相对静态的系统描述，看看各种类和对象是否真正有意义。在程序设计和测试中，这些用例也成为测试实例的源泉。在这种方式中，一组用例为观察所开发的系统提供了另一种角度，可以作为一种实在的检查。

用例将系统看做一种（动态的）工作的实体。因此，它们有可能诱使设计者从功能性的角度去观察这个系统，并使他们偏离了找出能映射到类的有用概念的本质性工作。特别是到了那些有着结构分析背景，且较少面向对象程序设计经验的人们手里，强调用例很容易引向某种功能分解。一组用例并不是一个设计，对于系统使用的关注也必须和与之互为补充的对系统结构的关注相配合。

一支开发队伍也可能被引诱到想去发现和描述所有的用例。这实际上并无益处，而且是一项代价高昂的作业。就像我们在为系统寻找候选类中的情况一样，在某个时候我们必须说，“够了就是够了，现在是去试一试我们已经找出的东西，看看会出现什么情况的时候了。”只有通过一组表面上能说得通的类和一组说得通的用例，我们才能在进一步的开发中获得反馈，这种反馈对于最终得到一个好系统是必不可少的。要确定何时应该结束一种有价值的活动确实很困难，而在这里，我们明明知道以后还会再转回来以便完成整个工作，确定何时结束这种活动就更困难了。

多少用例就足够了？一般说，这个问题不可能有答案。当然，对于一个特定项目，会有一个时刻，那时我们很清楚已经覆盖了系统的大部分常规功能，更多不那么平常的情况以及错误处理问题中的相当一部分也有了处理。这就是进入下一轮设计和编程的时候了。

当你试图评价一组用例对系统的覆盖情况时，将它们划分为主要用例和次要用例常常很

起作用。主要用例描述了系统的最常见的和“正常的”活动，次要者描述那些不那么正常的或者出错时的情景。次要用例的一个实例是上面“打电话”的一个变形，其中在拿起话筒后拨的是自己的号码。人们常说，如果已经覆盖了80%的主要用例和某些次要用例，这就是前进的时候了。当然，因为我们并不知道到底哪些东西组成了“所有的用例”，所以这只能是一条经验规则。实际经验和明智的判断将能在这里起作用。

这里所提到的概念、操作和关系都是我们从对应用领域的理解中自然产生的，或者是在对类结构的进一步工作中浮现出来的。它们代表了对应用的理解，经常是对基本概念的一种分类方式。例如，云梯车是一种救火车，救火车是一种卡车，卡车是一种车辆。23.4.3.2节和23.4.5节从取得进展的角度解释了观察类和类层次结构的几种方式。

注意视觉图形工程。在某个阶段，你将被要求将设计展示给某些人，你将做出一组图形，以解释将要构造的系统的结构。这将是一种极其有益的演习，因为它能帮助你注意力集中到系统最主要的方面，要求你用其他人能够理解的方式去表述你的想法。做报告是一种最宝贵的设计工具。为那些有兴趣并能提出建设性批评意见的人准备一个意在使他们能真正理解的报告，这是一个将自己的想法概念化并清晰地表达出来的很好练习。

然而，有关设计的正式报告也是一种很危险的活动，因为存在着很强的诱惑力，使人想去描述一个理想的系统（一个你盼望自己能构造出的系统，一个你的高级管理层盼望他们能够拥有的系统），而不是你所有的或者你可能在合理时间内生产出来的系统。当存在着相互竞争者，而且行政部门并不真正理解或者不关心“细节”时，报告也可能变成一种编造谎言的比赛。开发队伍在这里展示其最浮夸的系统，以便保住自己的工作。在这种情况下，代替思想的清晰表述的常常是大量莫名其妙的难懂辞藻和缩略语。如果你是这种报告的听众，特别当你是决策者或者资源的控制者时，绝对重要的就是你应该从实际计划中区分出哪些仅仅是意愿性的想法。高质量的报告材料并不能保证所描述的系统也有高的质量。事实上，我常常看到一些真正关心实际问题的组织，在与那些较少关心实际产出系统的组织竞争中，在展示其结果时却很快就落入了陷阱。

在寻找能够用类表述的概念时，应该注意到，系统里的某些重要性质是不可能用类表示的。例如，可靠性、性能和可测试性都是系统里能实测的重要性质，然而，即使是最彻底的面向对象系统也无法将其可靠性局部化为一个可靠性对象。系统里的这种弥漫性的属性只能描述并去为它们而进行设计，最终通过实测进行验证。对这些性质的关注必须应用于所有的类中，可能需要反应在一些类和组件的设计与实现规则中（23.4.3节）。

23.4.3.2 步骤2：描述操作

精化有关的类，描述它们的一组操作。很自然，不可能将找类的工作与确定它们所需操作的工作截然分开。然而，这里也存在着一种实际的区分，确定类时的注目点在于关键性的概念，有意淡化了类的计算性质；而在描述操作时，关心的就是找出一组完整可用的操作。同时考虑这两方面的问题常常会因为太困难而没法做，特别是因为许多相关的类还需要一起设计。在需要同时考虑这些问题的时候，CRC卡片常常很有帮助（23.4.3.1节）。

在考虑需要提供哪些函数时，存在几种不同的哲学。我建议采用如下策略：

- [1] 考虑这个类的对象应该如何构造、复制（如果允许）和析构。
- [2] 定义由该类所代表的概念所需要的最小操作集合。典型情况是将它们作为成员函数（10.3节）。

[3] 考虑为了记述的方便需要增加哪些操作，只将其中最重要的几个包括进来。这些操作常常成为非成员的“协助函数”（10.3.2节）。

[4] 考虑哪些操作应该是虚的，也就是确定那些将这个类作为界面，应该由派生类提供实现的函数。

[5] 对于本组件中所有的类，通盘考虑它们在命名方面和功能方面可能取得哪些共性。

这很显然是一种最小化的说法。把想像中可能有用的函数都加进去，并把所有函数都做成虚的将容易许多。但是，函数越多，其中的某些函数始终都不使用的可能性也就越大，它们也更容易限制随后的实现以及系统将来的演化。特别是那些直接读写对象状态中某些部分的函数，它们常常会将类束缚于某种单一的实现策略，从而严重地限制重新设计的可能性。这种函数降低了抽象的层次，使之脱离了被实现的概念。增加函数也增加了实现者的工作——还有重新设计时的设计师的工作。与某个函数已变成一种责任之后再去删除它相比，在某种需要已经弄清楚后加入一个函数，做起来要容易得多。

这里要求明确地做出将函数作为虚函数的决策，而没有把这件事情作为默认规定或者实现细节。提出这种要求的原因是，将一个函数做成虚函数，将对其所在类的使用以及这个类与其他类的关系产生至关重要的影响。即使在某个类里只有一个虚函数，该类的对象的布局也不会像C或者Fortran语言的对象那样简单了。还有，如果某个类有了一个虚函数，它也就有潜力作为一些尚未实现的类的界面，而这个虚函数也就隐含着对那些尚未实现的类的依赖性（24.3.2.1节）。

注意，这种最小化要求设计师做更多的工作，而不是更少。

在选择函数时，最重要的就是集中关心它应该做什么，而不是去关心它应该怎样做。也就是说，我们应该更多地注意所需要的行为，而不是去关注实现中的问题。

按照函数对于对象内部状态的使用情况将它们分类，有时也很有用处：

- 基础操作：构造函数、析构函数和复制操作
- 探查操作：那些不改变对象状态的操作。
- 修改操作：修改对象内部状态的操作。
- 转换操作：那些基于操作所针对的对象的值（状态），产生其他类型的对象的操作。
- 迭代器：访问或使用所包容的一系列对象的操作。

这些分类并不是互不相关的。例如，一个迭代器也可以设计为一个探查操作或者修改操作。这些类比也就是一种分类方式，可以帮助人们处理类界面的设计问题。很自然，同样可以有其他分类方式。这种分类方式在维持一个组件内部各个类间的统一性方面特别有用。

C++ 通过`const`和非`const`成员函数的方式支持对探查操作和修改操作的划分。与此类似，这里还直接支持构造函数、析构函数、复制操作和转换函数的概念。

23.4.3.3 步骤3：描述依赖性

精化有关的类，描述它们之间的依赖关系。24.3节里讨论了各种各样的依赖关系。在设计环节中，特别应该考虑的是参数化、继承关系和使用关系。这里的每种关系都涉及到有关一个类在为系统里某单一性质负责的意义方面的考虑。担负起明确的责任并不意味着这个类本身就要保存所有的数据，也不意味着它的成员函数需要直接完成所有必须的操作。与此相反，每个只需占有责任中的一片领地的类，应保证这个类完成的大部分工作是通过直接请求“其他地方”，由另一些以有关的子工作为己任的类去处理。当然也要当心，这种技术的过度使用

可能导致低效率和无法理解的设计，可能做出大量的类和对象，以至于它们什么都不做，只去产生瀑布式向前传送的服务请求。如果某件事情现在可以在这里做，就应该在这里完成。

在设计阶段（而不是在实现阶段）就需要考虑继承关系和使用关系，这一需求直接源于采用类来表示概念。它也意味着设计的单位应该是组件（23.4.3、24.4节），而不是类。

参数化（常常引向使用模板）是一种将隐式的依赖关系显式化的方法，这样可以同时表述几种选择，而不必增加新概念。经常有这样的选择：是把某种东西留做对环境的依赖关系，将它表述为继承树上的一个分支，还是采用一个参数（24.4.1节）。

23.4.3.4 步骤4：描述界面

描述界面。私用函数通常不必在设计阶段考虑。在设计阶段必须考虑的实现问题最好是作为关于依赖性的步骤2中的一部分进行处理。说得更强一些，我使用一条经验规则：对于一个类而言，除非它存在着至少两种有显著差异的实现方式，否则这里大概就有些问题。也就是说，这可能是一个伪装成类的具体实现，而不是某个真实概念的代表。在许多情况下，考虑对于一个类是否可能做某种形式的延迟求值，是回答下面问题的一种很好方式：“这个类的界面是与实现无关的吗？”

请注意，公用基类和友元也是一个类的界面的一部分，参见11.5节和24.4.2节。通过分别定义保护界面和公用界面，为继承和普通客户提供分离的界面，这种工作会有很好的回报。

正是在这个步骤中，应该考虑和描述参数的确切类型。这里的理想应该是使尽可能多的界面能利用应用层的类型静态确定。参见24.2.3节和24.4.2节。

在描述界面时，也应该为这些类向外看一看，是不是某些操作支持了多于一个的抽象层次。举个例子，类`File`的某些成员函数可能有类型为`File_descriptor`类的参数，它的另一些函数可能以表示文件名的字符串为参数。`File_descriptor`的操作与文件名操作并不在同一个抽象层次里，所以就应该怀疑它们是否应该在同一个类中。或许最好是有两个文件类，一个支持文件描述符的概念，另一个支持文件名的概念。在典型情况下，一个类里的所有操作都应该支持同一个抽象层次。如果它们不是这样的，那么就应该考虑重新组织这个类及其相关的类。

23.4.3.5 类层次结构的重组

在步骤1和步骤3里，我们都检查了类和类层次结构，看它们是否适合我们的需要。典型情况是并不适合，因此我们就必须去重新组织，改进其结构，改进设计或实现。

最常见的对类层次结构的重组是将两个类中公共的部分提取出来形成一个新的类，还有就是将一个类分裂为两个。在这两种情况下，结果都是三个类：一个基类和两个派生类。什么时候应该去做这种重组？什么情况能说明这种重组可能有价值呢？

不幸的是，对于这些问题都不存在简单而通用的回答。这当然不会令人感到意外，因为这里谈论的不是那种小的实现细节，而是修改一个系统的结构。最基本也很不容易做的操作就是去寻找类之间的共性，提取出它们共同的部分。有关共性的确切准则是无法定义的，但它应反应了系统中一些概念之间的共性，而不只是实现的方便。也存在一些关于两个或多个类具有可能提取出的共性的线索，如公共的使用模式、类似的操作集合、类似的实现，还有就是在设计讨论中这些类常常一起出现。在另一个方向上，如果一个类的操作的子集合具有差异显著的使用模式，或者这种子集合访问的总是表示之中不同的子集合，或者这个类常常出现在相互无关的讨论之中，就说明这个类可能是分裂为两个的候选者。有时，将一组相关的类做成一个模板也是系统地提供一组必要功能的方式（24.4.1节）。

因为类和概念之间的紧密关系，与类层次结构组织有关的问题常常表现为类的命名问题，或者有关设计的讨论中对类名字的使用问题。如果在设计讨论中所用的类名字，或者由类层次结构所蕴涵的分类方式听起来就很别扭，那么这里很可能有改进类层次结构的机会。注意，我说的意思也包括两个人分析类层次结构比一个人好得多。如果你正好找不到其他人一起讨论设计，那么可以写一个有关设计中类名字使用的讲稿，这也是一种很有帮助的做法。

设计中最重要目标之一就是提供一个界面，使之能够在变化之上保持稳定（23.4.2节）。做到这点的最好方式是将一个被许多类所依赖的类做成抽象类，其中只给出最一般的操作。最好将细节留给更特殊的派生类，因为直接依赖它们的类和函数都会少一些。应强调的是：依赖于某个类的类越多，这个类就应该越一般，它所揭示的细节也应该越少。

存在着一种很强的诱惑力，推动人们把更多的函数（和数据）放进被许多地方使用的类里，这常常被看做是一种使类更有用、更少需要（进一步）改变的方法。这种想法的效果就是做出带着庞大界面的（24.4.3节）和由一些相互无关的函数使用的数据成员的类。而这又隐含着，只要这个类所支持的许多类之一需要有明显的变化，这个类本身也必须跟着修改。这一情况转而又使这些变化影响到许多并无关系的用户类和派生类。对于那种处于设计中心的类，我们不应该使它复杂化，而应该保持它的一般性和抽象性。在需要时，特定的功能应当通过派生类提供，参见 [Martin, 1995] 里的例子。

按照这种方式思考将导致一种抽象类的层次结构，其中接近根部的是最一般的类，大部分其他类和函数都依赖于它们。在树叶处的是最特殊的类，只有很少的代码片段直接依赖于它们。作为例子，请考虑 *Ival_box* 层次结构（12.4.3节、12.4.4节）的最后版本。

23.4.3.6 模型的使用

在我准备写一篇文章时，总会试着去找一个可以参照的模型。也就是说，不是立即投入打字，而是去找有关类似论题的文章，看看能不能找到一篇可以作为我的文章的初始模式的文章。如果选定的模型就是自己原来写的针对相关论题的文章，我甚至可以让原文字的某些部分留在那里，根据需要修改其他部分，只在那些我希望传达的信息在逻辑上需要的地方加入新信息。例如，这本书就是基于它的第1版和第2版写出来的。这种写作技术的一个极端是格式信件，对于那类情况，我只是简单地填入姓名，可能再加上几行，使信件“个性化”。从本质上看，我写这种信件的方式就是只描述与模型的差异。

这样为新的设计使用现存的系统作为模型的用法在所有的创造性活动里都是正常情况，而不是例外。只要可能，就应该在前面工作的基础上设计和编程。这样做限制了设计师必须处理的问题的自由度，使人一下就可以将注意力集中到不多的几个问题上。让某个重要项目“完全从空白开始”可能是令人兴奋的，然而，一个更精确的描述常常是“醉人的”，结果是在不同设计选择间的一种醉鬼式的徘徊。有一个模型并不是限制，也不是要求奴隶式地追随这个模型，它只是使设计师可以比较自由地一次考虑系统的一个方面。

请注意，模型的使用是无价之宝，因为任何设计都是其设计师经验的集成。采用一个明确的模型将使模型的选择成为一项有意识的决策，使许多假设明确化，定义了一组公共词汇，为设计提供了一个初始框架，并增加了设计师们采用公共途径的可能性。

很自然，初始模型的选择本身就是一个重要的设计决策，常常只有在查询了许多潜在模型并评价了各种替代方式之后才能决定下来。进一步说，在许多情况下，只有在理解了使一个模型适应于特定新应用的想法而必须做的主要修改之时，才能说它是合适的。软件设计很

困难, 我们需要取得所有可能的帮助, 我们不应该拒绝使用模型, 不应以这种方式避免“模仿”的蔑视语, 那完全是用错了地方。模仿是最真挚的恭维形式, 而利用模型和以前的工作作为启示则是(在礼节和版权法律的范围之内)在任何领域中开展创新性工作的一种可以接受的技术: 对莎士比亚足够好的东西对我们也一样。有些人把在设计中利用模型称为“设计的重用”。

将出现在许多设计中的一般性元素和某些有关它们所解决的设计问题, 以及使用它们的条件描述一起建档是一个很明显的想法——至少你想到它之后是如此。词语模式常被用于描述这种一般性的有用的设计元素, 存在着一些有关建档各种模式及其使用的文献(例如, [Gamma, 1994] 和 [Coplien, 1995])。

作为设计师, 去了解在某个特定应用领域中的流行模式, 这也是一个很好的想法。作为程序员, 我喜欢那些带有一些作为具体示例的代码的模式。与大部分人一样, 在我有了具体示例(在这里是阐释模式使用的一段代码)的帮助之后, 我才能更好地理解一种一般性的思想。大量使用模式的人有一套特殊词汇, 这些词汇使他们之间的交流更容易进行。不幸的是, 这也有可能变成一种私密性的语言, 有效地防止外人的理解。与往常一样, 保证参与一个项目中不同部分的人之间的有效通信是至关重要的(23.3节), 对于大的设计和编程团体而言也是如此。

每个成功的大型系统都是某个更小一些的正在工作的系统的重新设计。我不知道有任何事实居于这一规律之外。我能想到的最接近那类情况的东西都是一些失败的项目, 一些花掉大笔费用而一塌糊涂许多年的项目, 还有一些在其预定结束日期的多年之后才成功的项目。这种项目都是无意识地(常常也是不公开承认地)先简单地做出一个不能工作的系统, 而后转到一个能工作的系统, 最后又经过重新设计, 使之成为一个接近初始目标的系统。这也意味着, 动手从空白开始构造一个大型系统是个傻念头, 正好符合最后原理。作为我们目标的系统越大、越雄心勃勃, 有一个可以开始工作的模型也就越重要。对于一个大型系统而言, 能实际接受的模型只有那些在某种意义上小一些的、与之相关的正在工作中的系统。

23.4.4 试验和分析

在开始一个雄心勃勃的开发项目时, 我们并不知道构造这个项目的最佳方式。经常是, 我们甚至不能确切地知道这个系统需要做什么, 因为只有通过构造、测试和使用这个系统的努力, 各种特殊情况才能变得更清晰。那么在还没有构造出完整的系统之前我们怎么才能取得一些信息, 设法理解哪些设计决策最为重要? 怎样去估计它们所蕴涵的后果呢?

我们通过试验。还有, 我们在有了一些可以分析的东西之后, 立即去分析有关的设计和实现。最经常也最重要的是, 我们需要讨论各种设计和实现可能性。除了极少的例外, 设计都是一种社会性活动, 设计在报告和讨论中逐步发展。黑板经常就是最重要的设计工具, 没有它, 有关设计的萌芽式概念就不能发展, 不能在设计师和程序员之间共享。

最流行的试验形式看来是构造出一个原型, 也就是系统的一个削减了规模的版本, 或者是系统的一部分。对于原型没有严格的性能标准, 机器和程序设计环境资源通常也比较富裕, 参与的设计师和程序员也都有很不平常的良好教育、经验和工作动力。这里的想法就是尽可能快地得到一个可以运行的系统版本, 以便能探索各种设计和实现选择。

如果做得好, 这种途径可以很成功。但它也可以成为懒散借口。问题在于, 原型的重

点很容易从“探索各种设计和实现选择”转移到“尽可能快地得到某种能运行的系统”。这又很容易引向不关心原型的内部结构（“不管怎么说，这不过是个原型罢了”），并轻视那些围绕着原型实现，使之更有用的设计工作。一个暗礁是，这样一个实现很可能堕落为最坏的一类吞噬资源的恶魔，在给出“几乎完全的”系统幻像的同时带来的是维护的噩梦。按照定义，原型几乎不必有内部结构、效率，不必具有使之能扩展为实际使用规模的能保持下去的基础结构。因此，一个“几乎是产品”的“原型”将用掉许多时间和资源，而这些时间和资源最好还是用到别的地方。对开发者和经理者的诱惑是将原型做成产品，并把“性能工程”留到下一次发布。如果误用这种方式的话，原型就会违背设计所追求的所有东西。

与此相关的问题是原型的开发者可能爱上他们的工具，以至忘记了作为产品的系统未必能负担起他们（所需要）那些便利条件的成本。由他们的小研究组提供的在约束条件和形式化方面的自由度，在一个面对着相互锁定的截止日期的更大开发组中也很难以维持。

另一方面，原型也可以极有价值。考虑一个用户界面的设计。在这种情况下，系统中不直接与用户交互部分的内部结构常常是无关紧要的，也没有其他可行方式能得到用户对系统观感的反应方面的经验。另一个实例是完全为研究一个系统的内部网络所设计的原型。在这里，用户界面非常简单，很可能采用模拟的用户，而不用真实用户。

做原型是一种做试验的方式。在构造原型时最希望得到的结果是在构造它的过程中获得的洞察力，而不是原型本身。对一个原型的最重要评判标准是，它应该是如此的不完全，以致很显然它是一个明显的试验性的媒介，不经过大范围的重新设计和重新实现就不可能转为产品。让原型“不完全”有助于将注意力集中在试验，也使原型变成产品的危险性减到最小。这样也使另一种诱惑减到最小程度，那就是试图将产品设计的基础尽可能地靠近原型设计——这样也会忘记或者轻视了原型的内在局限性。在利用之后，就应该将原型丢掉。

应该记住，在许多情况下，存在着许多可以代替原型的试验技术。在能够使用它们时，最好是使用它们，因为它们更严格得多，而且对设计师的时间要求、对系统的资源要求都更少。这方面的例子如数学模型和各种形式的模拟器。实际上可以看到一种连续系，从数学模型，穿过牵涉到越来越多细节的模拟器，再到原型，到部分实现，直至完整的系统。

这一情况也会引起一种想法：让系统通过一系列的重新设计和重新实现，从一个初始设计和实现开始成长。这是一种很理想的策略，但它可能对于设计和实现工具提出极高的要求。还有，这种途径也有一种危险，它可能受到反应了初始设计决策的大量代码的拖累，以至更好的设计根本无法实现。至少在目前，这一策略看来还只限于用在小型到中型的项目，在其中不大会出现对整体设计的重要修改；还有就是在系统的初始发布之后的重新设计和重新实现，因为在那里，这样的策略是不可避免的。

除了通过试验性设计提供有关各种设计选择的洞察力之外，对设计和/或实现本身的分析也是获取深入认识的重要源泉。举例来说，研究类之间的各种依赖关系（24.3节）可能极有帮助，传统的实现工具，如调用图、性能实测等，也都不应小看。

注意，规范说明（规格说明，分析阶段的输出）和设计也像实现一样，存在着许多错误。事实上，它们中的错误可能更多，因为它们更不具体，其描述常常更不精确，而且不能执行，在典型情况下也没有足够精妙的工具的支持，至少不像在对实现进行检查和分析时可用的工具那么好。增强在表述设计时所用的语言或者记法的规范性，可以向着利用工具帮助设计师的方向有所前进。但在这样做时，绝不应削弱了实现中所用的程序设计语言（24.3.1节）。还有，一

种形式化记法本身也会成为复杂性和问题的根源。当某种形式记法并不适合它所面对的实际问题，形式化的严格性超过了参与工作的设计师和程序员的数学背景和熟练程度，以及一个系统的形式化描述与它假定应该描述的系统脱节的时候，就会出现这种情况。

从本质上说，设计就是一种很容易出错的且难于用有效工具支持的活动。这些都使经验和反馈成为不可或缺的。因此，将软件开发看成是一种线性的过程，从分析开始到测试结束，这种看法存在着根本性的缺陷。需要强调反复进行的设计和实现，以便从开发的各个不同阶段所取得的经验中获得反馈。

23.4.5 测试

没有完成测试的程序不能工作。设计和/或验证一个程序，使之能第一次就工作得很理想是难以实现的，除了对最简单的程序之外。我们应该向着这个理想努力，但我们绝不应愚蠢到认为测试很容易的地步。

“怎样测试？”这又是一个无法给出一般性答案的问题。然而，“何时测试”则有一个一般性的回答：尽可能早做，尽可能经常去做。测试策略应该作为设计和实现工作的一部分产生出来，或者至少应该与它们平行地开发。一旦有了一个能够运行的系统，就应该开始测试。将认真地测试推迟到“整个实现完成之后”就是要求推翻计划安排或发布有缺陷的产品。

在所有可能之处，都应该特别考虑如何设计系统，使测试工作相对而言比较容易进行。特别是常常应该把为测试服务的机制直接设计到系统里。有时没有这样做是因为害怕造成代价高昂的运行时检查，或者担心一致性检查所需的冗余可能形成急剧扩大的数据结构。这种担心通常是放错了地方，因为只要有必要，大部分测试代码和冗余都可以在系统发布之前从代码中剥掉。在这方面，断言（24.3.7.2节）有时很有用处。

与特定的测试相比，更重要的是一种思想，系统的结构应该使我们有一种合理的机会去使我们和我们的用户/顾客确信：我们能通过静态检查、静态分析和测试的组合清除掉各种错误。在开发出了某种容错策略的地方（14.9节），也应该将测试策略设计为一种补充，并使之尽可能与整体设计中的有关方面靠近。

确定需要做多少测试常常也非常困难。当然，最常见的毛病是测试不足，而不是测试过度。在与设计和实现相比，到底需要将多少资源分配给测试，这个问题自然依赖于系统的性质和构造它所用的方法。不过，作为一条经验规则，我要建议在时间、工作量和智力方面，分配给系统测试的资源应该多于构造其初始实现的资源。测试应当集中于那些可能造成灾难性后果的问题，以及那些可能频繁发生的问题。

23.4.6 软件维护

“软件维护”是一种用词不当。“维护”这个词会引起一种与硬件类似的误解。软件不需要加油，没有会磨损的运动部件，不存在裂隙使水可以汇集其中而引起锈蚀。软件可以一模一样地复制，在分秒之间传送过很长的距离。软件不是硬件。

在软件维护名目下的活动实际上是重新设计和重新实现，因此属于普通的程序开发循环。在设计中强调灵活性、可扩展性和可移植性之时，也就是直接针对传统上软件维护所要解决的问题。

与测试一样，维护也不能是一种事后的思考，不能是一种游离于开发主流之外的活动。

特别重要的是,项目开发的人员组织应该具有某种连续性,如果与原设计师和程序员之间不存在某种联系,将维护问题成功地传给新的(通常更缺乏经验的)一组人是非常困难的。如果必须有很大的人员变动,那么就需要强调向新人转移有关系统结构和系统目标的理解。如果将“维护组”放在某种位置,让他们去揣测系统的体系结构,或者仅仅根据实现去推断系统部件的用途,那么,在局部补丁的不断冲击之下,系统的结构就会很快恶化。通常文档往往更注重谈论细节,而不是帮助新人理解关键性的思想和原理。

23.4.7 效率

Donald Knuth认为“不成熟的优化是一切罪恶之源”。有些人在这方面学得实在太好了,以致于将一切对效率的关心都看做是罪恶。与此相反,在整个设计和实现工作中,都应该将效率问题放在心上。当然,这并不意味着设计者应该关心细微的效率问题,但一级的效率问题是必须考虑的。

处理效率问题的最佳策略就是产生出一个清晰简单的设计。只有这样的设计,才能在项目的整个生存期间,既能够作为性能调整的基础,同时又保持相对的稳定性。避免庞大化极其重要,这种情况是大型项目的“黑死病”。人们过分经常地加入“特殊需要”的特征(23.4.3.2节、23.5.3节),为支持这种虚饰,最终导致两倍或者四倍的系统规模和运行时间。更糟糕的是,常常很难对这种过分费神做出的系统做分析,因为难以区分那些不可避免的开销和可能避免的开销。这样,甚至基本的分析和优化也无法进行。优化应该是细致分析和性能实测的结果,而不能随机地摆弄代码。特别是在大型系统里,设计师或者程序员的“直觉”对于指导与性能有关的事项而言都是极不可靠的。

最重要的是避免那些在本质上就是低效的结构,避免那些需要花费过多的时间和聪明才智才可能将其优化到可以接受的性能水平的结构。类似地,尽量少用那种具有内在的不可移植的结构和工具也非常重要,因为使用这种结构也就是宣告该项目将要运行在老的(性能较差或更昂贵的)计算机上。

23.5 管理

只要有一点点能说得通,大部分人都会去做他们被要求去做的事情。特别是,如果在—项软件项目的环境里,按照某种方式工作会得到奖赏,否则将受到惩罚,那么只有最出色的设计师和程序员才会面对着管理层的反对、冷漠和官僚作风,冒着职业风险去做他们认为正确的东西^①。这也就意味着,一个组织应该有一种与它所陈述的设计与编程目标相匹配的回报结构。然而,并非如此的情况太常见了:程序设计风格的重要改变只能通过相应的设计风格的变化,而这两者通常又要求管理风格的变化才能生效。思想和组织的惯性都太容易导致局部的改变,而它又无法得到使其成功所需要的全局性改变的支持。一个相当典型的例子就是转到某种支持面向对象程序设计的语言,例如C++,但却没有与此同时改变相适应的设计策略,以利用这种语言的功能(另见24.2节)。另一个例子是改为“面向对象的设计”,但却不引进语言去支持它。

^① 如果一个管理机构将程序员当作白痴,它很快就会有—批愿意做白痴并只能按照白痴的方式做事的程序员。

23.5.1 重用

代码和设计的重用经常被用于作为采纳某种新程序设计语言或设计策略的原因。然而，许多组织却奖励那些选择去重新发明车轮的个人和小组。举例来说，程序员的生产能力可能是按照代码行数评价的，他会愿意牺牲收入和可能的地位去写基于标准库的小程序吗？管理员的报酬可能与她的小组里的人数有某种比例关系，在她可能在自己的组里雇佣另外一些人手的情况下，她会去采用其他小组的软件产品吗？一个公司可能获得了一项政府合同，在其利润与开发费用之间有着固定的百分比，这时公司会考虑去使用最有效的开发工具，以便使自己的利润最小化吗？奖励重用非常困难，但是，除非管理层能够找到一些方式去鼓励和奖赏重用，否则它就不会发生。

重用是一种社会现象，我可以利用其他人的软件，要求是：

- [1] 它能工作：为了能重用，软件必须首先是可用的。
- [2] 它可以理解：程序结构、注释、文档和教学材料都非常重要。
- [3] 它能与并不是为了与之共存而专门写出的软件共存。
- [4] 它有支持（或者我愿意自己支持自己；一般情况下并不是这样）。
- [5] 它是经济的（我能与其他用户共同分担开发和维护费用吗？）。
- [6] 我能够找到它。

在此之上，我们还可以加上：在有人已经“重用”了某个组件之前，它都不能算是可重用的。将组件纳入其环境的工作通常要求去精化其操作，推广其行为，改进其功能，以便与其他软件共存。直到这种事情至少已经做过一次之后，我们才能说它可以重用了，因为，即使是在极其小心地设计和实现出的组件里，通常也有一些没想到的不希望有的粗糙角落。

我的经验是，使重用得以存在的必备条件是有人将它作为自己的事情，去为这种共享而工作。在一个小组里，这通常意味着有某个个人，无论是由于指定还是由于偶然，变成了公共库和文档的管理人。在更大的组织中，这意味着要有一个小组或者部门专门去收集、构造、写文档、推广和维护其他许多小组所使用的软件。

这样一个“标准组件”小组的重要性无论怎样估计都不过分。注意，作为一级近似，一个系统反应了生产它的组织的情况。如果一个组织中没有某种机制去推动和奖励合作与共享，那么合作与共享必定是罕见的。标准组件组必须自动地去选出自己的组件，这也意味着好的传统文档是必需品，但这还不够。在此之外，这个组件小组还必须提供教材和其他信息，使潜在的用户能够找到某个组件，并理解为什么它可能有用。这也意味着组件小组必须着手去做一些在传统上与市场营销和教育相关联的活动。

任何时候只要可能，这个组的成员都应该与构造应用的人员密切合作。只有这样，他们才能充分理解用户的需求，察觉到在不同应用之间共享组件的可能性。这也表明了这种组织方式有一种咨询作用，表明可以利用这种关系将信息传入或者传出这个组件小组。

这种“组件小组”的成功需要依据其客户的成功情况进行评价。如果只是简单去评价它说服开发组织接受的工具和服务量，这种小组就会堕落为一种叫卖商品软件的小贩，仅仅是去鼓吹不断变化的时尚。

并不是所有代码都需要重用，可重用性也不是一种具有普遍性的特征。说一个组件是“可重用的”，就意味着它可以在某个确定的框架里，只要求做较少工作或者无需任何工作就

可以重用。在大部分情况下,转移到不同的框架里需要做大量工作。在这一方面,重用的情况很像可移植性。最重要的是应注意到,重用是将设计目标定位于重用,基于经验去精化组件,以及有意搜寻可能重用的现存部件而取得的结果。重用不会从漫不经心地使用某些特定语言特征或者编码技术中魔术般地冒出来。C++ 的许多特征(如类、虚函数和模板)都使我们可以适当地表述设计,使重用变得更容易些(并因此更可能出现),但是这些特征本身并不保证可重用性。

23.5.2 规模

个人或者组织都很容易因为“正确行事”而感到兴奋。在机构的意义下,这常常可以翻译为“取得进展并且严格按照正确程序进行着”。在这两种情况下,在真诚的、常常是强烈的改进做事方式的愿望中,最早的牺牲品可能就是常识。不幸的是,一旦丢掉了常识,可能在不知不觉中做出的坏事也就没有了限度。

考虑在23.4节列出的开发过程中的各个阶段,以及在23.4.3节列出的设计步骤中的各个阶段。不难进一步将这些阶段加工成一种更完善的设计方法,对其中的每个阶段都给以更精确的定义,带有定义良好的输入和输出,并采用某种半形式化的记法去表述输入和输出。可以开发出一些检查表来保证遵守这种设计方法,开发出一些工具来强制性地实施过程性的和记法性的规范。再进一步,查看在24.3节所展示的分类方式和依赖关系,人们也可以判决出某些依赖关系就是好的而其他则是坏的,并提供一些分析工具来保证这种价值取向能在整个项目中统一地实施。为使这种“严谨的”软件开发过程更加完善,人们还可以定义文档的标准(包括拼写规则、语法和打字规范),代码的一般表述标准(包括有关允许或不允许使用的语言特征的规范,有关允许或不允许使用哪些种类的库的规范,有关代码缩排以及函数、变量和类型命名的规定等)。

这些中的大部分对于一个项目的成功都可能有帮助。至少说,如果要着手一个最终可能包含上千万行的代码、由数百个人参与开发、数千人在十年或更长的时期中维护和支持的系统的设计,如果没有类似上面所描述的这样有着良好定义的有点严格的框架,那就是一件愚蠢的事情。

幸运的是,大部分系统并不属于这样一类东西。然而,一旦接受了这样一种思想,认为这种设计方法或者坚持这样一组编码和文档标准才是“正确方式”的思想,强制性地要求将它应用到所有的地方和每一个细节,这样做,对于小项目就可能造成很荒唐的束缚和额外的开销。特别是它可能导致以倒腾纸片、填写表格作为衡量进展和成功的标准,以这些取代生产性的工作。如果出现这种情况,设计师和程序员将会离开这种项目,取而代之的则是一些官僚。

一旦某种(原本完全合理的)设计方法的滑稽可笑的误用在某个团体中出现,其失败又会成为在开发过程中避免所有规范方式的口实,这将很自然地带来一类混乱和失败,而设计出这种设计方法原本就是为了防止这些情况出现。

实际问题是,应当为特定项目的开发选择适当的规范性水平。不能期望对此问题有一个简单回答。所有的方法基本上都能对付小的项目,更糟糕的是,看起来差不多每种方法也都能用于大型项目,无论它如何病态地鼓励奇想,或者对所涉及的人有多么严酷,只要你愿意将大量的时间和金钱投入到这个问题中。

在每个软件项目中,最关键的问题就是如何维持设计的完整性。这一问题与规模增长的

关系比线性增长更快些。只可能有一个个人或者一小组人能够抓住并保持着对一个重要项目的整体目标的认识。大部分人必须将他们大量的时间用在子项目、技术细节、日常管理等上面,所以就很容易忘记总体设计目标,或更重视自己的局部和当前目标。不让每个个人或小组都有明确的任务去维护设计的完整性是一种导向失败的方法,不让每个个人或小组在作为整体的项目上付诸努力也是一种导向失败的方法。

对于一个项目或一个组织而言,缺乏一种长远目标比缺少某种孤立性质的危害性大得多。应该有一小群人去做这种工作,形成这样一种整体目标,将这一目标牢记在心,写出关键性的整体设计文档,写出对关键性概念的介绍,并一般性地帮助其他人将这一目标牢记在心里。

23.5.3 个人

采用如这里所描述的设计,将给熟练的设计师和程序员送上了一份超值礼券。同时,它也会使设计师和程序员的选择成为一个组织成功的最关键因素。

管理者常常忘记组织是由个人组成的,有一种流行的概念说:程序员都是一样的,可以互换的。这纯属谬论,它可以通过逐走大量最有成效的个人,并判决留下的人员应该在某种低于他们潜能的水平上工作,从而毁灭一个组织。说个人可以互换,条件就是不允许他们利用自己高于为完成工作所需的最低要求的那一部分能力。所以,虚构的互换性是非人性的,具有内在的非经济性。

大部分程序设计能力的评价方式都鼓励不经济的实践活动,没有考虑到关键性个人的贡献。最明显的例子是,用生产的代码行数、生产的文档页面数、通过的测试个数等评价进展,这些都采用得相当广泛。这种数字在管理层的图表上看起来很漂亮,但它与现实间的关系却是最贫乏无力的。举例来说,如果用生产出的代码行数来衡量生产率,重用的成功实施将表现为对程序员功效的一种否定。在重新设计软件的重要部分时,最佳原则的成功实施也会带来与此类似的影响。

评价工作产出的质量远比衡量产出的数量困难得多,然而,无论个人还是小组,都应该依据其工作质量而不是粗糙的数量测度给予回报。不幸的是,实用的质量测量体系的设计(按照我的了解)很难开始。此外,不能完全描述项目状态的度量方式将束缚发展。人们将调整自己去适应局部的时限,并按照度量方式的定义去优化个人和小组的功效。作为一种直接结果,这些将使整个系统的完整性和性能都受到损害。例如,如果用已删除的程序错误数或者已知遗留错误数来定义时限,我们就会看到,为满足这种时限,付出的代价是运行性能或者运行这个系统所需要的硬件资源。相反,如果只衡量运行性能,在开发者们针对标准测试优化性能的战斗中,错误率必然会上升。缺乏好的容易理解的质量测量体系,对管理者提出了极强的技术性专业知识方面的要求,而替代这些的只能是奖赏随机性行为而非实际进展的系统倾向性。不要忘记管理者也是人,管理者至少需要在新技术方面受到与他们所管理的那些人同样的教育。

与软件开发的其它领域一样,我们必须有更长远的考虑。基于一年的工作去评价一个个人的成效,从本质上说是不可能的。当然,大部分个人都有一致的长期记录,它可以作为技术评价的可靠预测器,对于评价其刚刚过去的工作也很有帮助。无视这种记录(在将个人仅仅看做在组织的车轮上可以互换的齿轮时正是如此),必将使管理者处于误导性的数量评价方式的可怜境地。

采取长期观点，避免“管理可互换的白痴学校”的一个推论就是，每个人（开发者和管理者）都需要较长时间才能成长到可以进入需要更高程度的技能的更有趣的工作中。这是对“履历发展”的跳槽和职位轮换的否定。关键性技术人员和关键性管理人员的低替换率必须成为一个目标，如果没有与关键性设计师和程序员的友好相处，没有最新的相关技术知识，管理者将不可能成功。在另一方面，如果没有胜任的管理者的支持，没有工作于其中的更大的非技术环境的最小程度的理解，设计师和程序员的小组也不可能成功。

在需要创新的地方，资深技术人员、分析师、设计师、程序员等，在引进新技术中扮演着最关键的也是非常困难的角色。正是这些人必须学习新技术，在许多情况下还需要抛弃老的习惯。这不是很容易的事情。这些个人通常都已经在采用老方式做事情方面付诸了大量的个人投入，也由于采用这些方式工作的成功而获得了在技术方面的声望。许多技术管理人员也是如此。

很自然，在这些个人中常有一种对转变的恐惧。这可能引起高估改变所涉及的各种问题，以及不愿意去正视采用老方式做事引起的问题。同样也很自然，为转变而呼喊的人们则倾向于高估新的做事方式的有利效用，低估由于改变而带来的问题。这两组个人必须交流，他们必须学会用同一种语言谈话，他们必须互相帮助去塑造出一个转变的模型。代替这种方式的一定是组织的瘫痪和两组人中最有能力者的流失。这两组人都应该记住，最成功的“老顽固”常常就是昨天的“年轻斗士”。有了并不蒙羞的学习机会，更有经验的程序员和设计师将能够成为转变的最成功、最有远见的拥护者。他们的健康的怀疑态度、有关用户的知识、对于组织性阻碍的熟识都是极其宝贵的。立即和彻底转变的拥护者也必须认识到，这样一个转变常常涉及到逐步地采纳新技术，过犹不及。相反，那些根本不希望转变的人们应该另找一个不需要变化的领域，而不是在一个新的需求早已显著地改变了成功条件的领域里，打一场气急败坏的后卫战。

23.5.4 混成设计

将做事情的新方法引入一个组织也可能是很痛苦的，该组织和组织中个人的分裂可能很明显。特别是，一夜之间的突然变化，将“老门派”中最有生产力的熟练成员变成“新门派”中最低效的新手，这通常是无法接受的。当然，不改变很难获得最大的收获，而重要的转变通常也有风险。

C++ 的设计就是希望将这种风险减少到最小，采用的方式是允许逐步采纳各种技术。虽然事情很清楚，使用C++ 获取最大的利益是通过数据抽象、面向对象的程序设计和面向对象的设计获得的；但不清楚的是，通过与过去彻底决裂是否能够最快地得到这些利益。这种清晰的决裂很少能够实行。更经常的是，对于进步的追求需要而且应该与有关如何控制这种转变的忧虑相调和。应考虑到：

- 设计师和程序员需要时间去获得新的技能。
- 新的代码需要与老代码合作。
- 老的代码需要维护（通常是无穷无尽的）。
- 在现存设计和程序上的工作需要（按时）完成。
- 需要将支持新技术的工具引进局部环境里。

这些因素会很自然地导致一种混成形式的设计——即使是在一些设计师的本意并非如此的地

方。人们很容易低估前面两点。

C++通过支持多种程序设计范型，以多种方式支持逐步引入组织中的观念：

- 程序员可以在学习C++ 的同时保持其生产能力。
- C++ 可以在缺乏工具的环境中产生显著效益。
- C++ 的程序片段可以与在C或者其他传统语言中写出的代码合作。
- C++ 有一个很大的与C兼容的子集。

这里的想法是，程序员可以把从传统语言移向C++的转变过程规划为：首先在采纳C++ 的同时保持传统的（过程式）程序设计风格，而后再使用数据抽象技术，最后，在已经掌握了这个语言及其相关的工具时，他们再转向面向对象的程序设计和通用型程序设计。注意，经过良好设计的库很容易使用，虽然它们的设计和实现要困难得多。因此，新手们在这一进步的早期阶段就可以从抽象机制的更高级应用中获益。

分步学习面向对象的设计、面向对象的程序设计和C++的思想，得到了C++中有关机制的支持，这些机制使C++代码能与采用不支持C++数据抽象和面向对象程序设计概念的语言写出的代码混在一起（24.2.1节）。可以让许多界面保持为过程式的，因为将任何事情搞得更复杂不会得到即时的利益。对于许多关键性的库而言，这些都已经由库的提供者完成了，所以程序员仍可以停在那里，忽略真正的实现语言。采用由C一类的语言写出的库是在C++里最早的也是在初期最重要的重用方式。

下一阶段（只在那些实际需要更精致的技术的地方）就是将用C或者Fortran一类语言写出的概念用类的方式提供，将数据结构和函数封装到C++的界面类中。11.12节中的字符串类就是将语义从过程加数据结构的层次提升到抽象数据结构层次的一个简单实例。在那里，采用对C字符串表示和标准C字符串函数的封装产生出一个字符串类型，它的使用就大大简化了。

类似技术可以用于将内部的或者单独的类型装入类层次结构中（23.5.1节）。这将使用其他语言写的代码也能出现在为C++所做的涉及到数据抽象和类层次结构的设计中也可以出现，而那些语言里没有上述概念，甚至存在着结果代码必须能从过程式语言里调用的限制。

23.6 带标注的参考文献

本章只是描绘了有关程序设计项目的设计问题和管理问题的表面情况。由于这一原因，在这里提供了一个很短的带标注的参考文献表。在 [Booch, 1994] 中可以找到一个更广泛的带标注的参考文献表。

[Anderson, 1990]

Bruce Anderson and Sanjiv Gossain: *An Iterative Model for Reusable Object-Oriented Software*. Proc. OOPSLA'90. Ottawa, Canada.

描述了重复式设计和重新设计模型，附带一个特殊实例和有关经验的讨论。

[Booch, 1994]

Grady Booch: *Object-Oriented Analysis and Design with Application*. Benjamin/Cummings, 1994. ISBN 0-8053-5340-2.

详尽地描述了设计，一种特殊的带有图形记法形式的设计方法，若干个在C++里表述的大型设计实例。该书更深入地讨论了本章中的许多问题。

[Booch, 1996]

Grady Booch: *Object Solutions*. Benjamin/Cummings. 1996. ISBN 0-8053-0594-7.

从管理的角度描述了面向对象系统的开发。包含范围广泛的C++代码实例。

[Brooks, 1982]

Fred Brooks: *The Mythical Man Month*. Addison-Wesley. 1982.

每个人每隔几年就应该再读一次本书。反对狂妄自大的训诫。在技术材料方面已经有点过时了,但在有关个人、组织和规模方面的内容则完全不过时。在1997年重印,ISBN 1-201-83595-9。

[Brooks, 1987]

Fred Brooks: *No Silver Bullet*. IEEE Computer, Vol.20, No. 4. April 1987.

有关大规模软件开发方法的一个综述,带有我们最需要的反对相信万能灵药(“Silver Bullet”,银弹)的训诫。

[Coplien, 1995]

James O. Coplien and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*. Addison-Wesley. 1995. ISBN 1-201-60734-4。

[DeMarco, 1987]

T. DeMarco and T. Lister: *Peopleware*. Dorset House Publishing Co. 1987.

少有的几本集中关注人在软件生产中的作用的著作。每个管理者的必读书。美妙得足以作为床头读物。是对许多愚蠢行为的解药。

[Gamma, 1994]

Eric Gamma et. al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2.

有关创建灵活的可重用软件的技术的分类目录,带有一个非平凡的、有着很好解释的实例。包含广泛的C++代码实例。

[Jacobson, 1992]

Ivar Jacobson et. al.: *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 0-201-54435-0.

透彻而实际地描述了在产业环境中的软件开发,特别强调用例(23.4.3.1节)。对C++做了很不合适的描述,就像它还是10年前的样子。

[Kerr, 1987]

Ron Kerr: *A Materialistic View of the Software “Engineering” Analogy*. In SIGPLAN Notices, March 1987.

本章和随后几章中所使用的类比在很大程度上应归功于这篇文章中的观点及其报告,以及此前与Ron的讨论。

[Liskov, 1987]

Barbara Liskov: *Data Abstraction and Hierarchy*. Proc. OOPSLA'87 (Addendum). Orlando, Florida.

讨论了继承的使用可能损害数据抽象。注意, C++有特殊的语言支持,以帮助避免这里所提出的问题(24.3.4节)。

[Martin, 1995]

Robert C. Martin: *Designing Object-Oriented C++ Applications Using the Booch Method*.

Prentice-Hall. 1995. ISBN 0-13-203837-4.

描述了怎样从问题出发, 沿着一条相当系统化的道路走到C++代码。比大部分有关设计的书籍更实际也更具体。包含广泛的C++代码实例。

[Meyer, 1988]

Bertrand Meyer: *Object Oriented Software Construction*. Prentice Hall. 1988.

第1-64页和第323-334页很好地介绍了面向对象编程和设计的一种观点, 带有许多有效的实际建议。本书的其他部分描述了Eiffel语言。存在着混淆Eiffel和一般性原理的倾向。

[Parkinson, 1957]

C. N. Parkinson: *Parkinson's Law and other Studies on Administration*. Houghton Mifflin. Boston. 1957.

有关管理过程可能导致的灾难的一个最好笑、最尖锐的描述。

[Shlaer, 1988]

S. Shlaer and S. J. Mellor: *Object-Oriented Systems Analysis and Object Lifecycles*. Yourdon Press. ISBN 0-13-629023-X和0-13-629940-7.

提出了一种有关分析、设计和编程的观点, 与我们这里所提的很不一样。有关讨论嵌在C++里, 在这样做时所用的一套词汇使它听起来与这里的讨论很类似。

[Snyder, 1986]

Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Proc. OOPSLA'86. Portland, Oregon.

可能是第一篇最好的有关封装和继承间相互关系的描述。还包括对多重继承中某些概念的很好的讨论。

[Wirfs-Brock, 1990]

Rebecca Wirfs-Brock, Brain Wilkerson and Lauren Wiener: *Designing Object-Oriented Software*. Prentice Hall. 1990.

描述了一种拟人式设计方法, 基于使用CRC卡片的角色扮演。其正文, 如果不是方法本身的话, 是基于Smalltalk的。

23.7 忠告

- [1] 知道你试图达到什么目的; 23.3节。
- [2] 心中牢记软件开发是一项人的活动; 23.2节、23.5.3节。
- [3] 用类比来证明是有意的欺骗; 23.2节。
- [4] 保持一个特定的实实在在的目标; 23.4节。
- [5] 不要试图用技术方式去解决社会问题; 23.4节。
- [6] 在设计和对待人员方面都应该有长期考虑; 23.4.1节、23.5.3节。
- [7] 对于什么程序在编码之前先行设计是有意义的, 在程序规模上并没有下限; 23.2节。
- [8] 设计过程应鼓励反馈; 23.4节。
- [9] 不要将做事情都当做取得了进展; 23.3节、23.4节。
- [10] 不要推广到超出了所需要的、你已有直接经验的和已经测试过的东西; 23.4.1节、23.4.2节。
- [11] 将概念表述为类; 23.4.2节、23.4.3.1节。

- [12] 系统里也存在一些不应该用类表述的性质；23.4.3.1节。
- [13] 将概念间的层次关系用类层次结构表示；23.4.3.1节
- [14] 主动到应用和实现中去寻找概念间的共性，将由此得到的一般性概念表示为基类；23.4.3.1节、23.4.3.5节。
- [15] 在其他领域中的分类方式未必适合作为应用中的继承模型的分类方式；23.4.3.1节。
- [16] 基于行为和不变式设计类层次结构；23.4.3.1节、23.4.3.5节、23.4.3.7.1节。
- [17] 考虑用例；23.4.3.1节。
- [18] 考虑使用CRC卡片；23.4.3.1节。
- [19] 用现存系统作为模型、灵感的源泉和出发点；23.4.3.6节。
- [20] 意识到视觉图形工程的重要性；23.4.3.1节。
- [21] 在原型成为负担时就抛弃它；23.4.4节。
- [22] 为变化而设计，将注意力集中到灵活性、可扩展性、可移植性和重用；23.4.2节。
- [23] 将注意力集中到组件设计；23.4.3节。
- [24] 让每个界面代表在一个抽象层次中的一个概念；23.4.3.1节。
- [25] 面向变化进行设计，以求得稳定性；23.4.2节。
- [26] 通过将广泛频繁使用的界面做得最小、最一般和抽象来使设计稳定；23.4.3.2、23.4.3.5节。
- [27] 保持尽可能小，不为“特殊需要”增加新特征；23.4.3.2节。
- [28] 总考虑类的其他表示方式。如果不可能有其他方式，这个类可能就没有代表某个清晰的概念；23.4.3.4节。
- [29] 反复评审、精化设计和实现；23.4节、23.4.3节。
- [30] 采用那些能用于调试，用于分析问题、设计和实现的最好工具；23.3节、23.4.1节、23.4.4节。
- [31] 尽早、尽可能频繁地进行试验、分析和测试；23.4.4节、23.4.5节。
- [32] 不要忘记效率；23.4.7节。
- [33] 保持某种适合项目规模的规范性水平；23.5.2节。
- [34] 保证有人负责项目的整体设计；23.5.2节。
- [35] 为可重用组件做文档、推介和提供支持；23.5.1节。
- [36] 将目标与细节一起写进文档里；23.4.6节。
- [37] 将为新开发者提供的教学材料作为文档的一部分；23.4.6节。
- [38] 鼓励设计、库和类的重用，并给予回报；23.5.1节。

第24章 设计和编程

使之保持简单，

尽可能简单，

但不要过于简单。

——阿尔伯特·爱因斯坦

设计和程序设计语言——类——继承——类型检查——程序设计——类代表什么——类层次结构——依赖性——包容——包容和继承——设计权衡——使用关系——编在程序里的关系——不变式——断言——封装——组件——模板——界面和实现——忠告

24.1 概述

本章从一般的角度并特别从C++的角度考虑程序设计语言对于设计的支持方式：

24.2节 类、类层次结构、类型检查和程序设计本身的基本角色。

24.3节 类和类层次结构的使用，将集中关注程序不同部分间的依赖关系。

24.4节 组件的概念（它是设计的基本单位）以及关于如何表述界面的一些实践性观点。更一般的设计问题已在第23章讨论，有关类的各种使用方式的细节将在第25章讨论。

24.2 设计和程序设计语言

如果我要架一座桥，我一定会认真地考虑用什么材料去建造它。还有，桥的设计也会受到材料选择的深刻影响，反过来也一样。石桥的合理设计不同于钢桥的合理设计，也不同于木桥的合理设计，如此等等。如果对各种材料及其使用一点也不了解，就不可能选到正确的建桥材料。当然，设计木桥并不要求你一定是个熟练的木匠，但你在做出用木材还是钢材作为建桥材料的选择时，必须理解木结构的基本性质。进一步说，虽然在设计一座木桥时，你个人不必是熟练的木匠，但你却需要有比木匠更多的有关木材性质的知识。

类似问题也出现在为某个软件选择一种程序设计语言的时候，你需要了解几种语言，要使所设计的软件片段取得成功，你需要有对所选的实现语言的相当细节的知识——即使你个人根本不去写有关软件中的一行代码。好的桥梁设计师特别注重材料的性质，并利用这些性质去提升自己的设计。与此类似，好的软件设计师也基于实现语言的威力去构造系统，并尽可能避免采用那些可能给实现者带来问题的使用方式。

有人可能认为，在只涉及到一个设计师/程序员的时候，这种对语言问题的敏感性是很自然的。然而，即使在这种情况下，程序员也可能错误地使用语言，由于不适当的经验而误入歧途，或者由于过度坚持在截然不同的语言上习得的程序设计风格。当设计师与程序员不同的时候（特别是如果他们并不共享同样的文化），把错误、生硬或者低效率引进系统中的可能

性几乎是确定无疑的。

那么，程序设计语言能对设计师起什么作用呢？它可以提供一些特征，使某些基本设计概念能直接在程序设计语言里表述。这将使实现更容易些，也使维护设计与实现之间的相互对应变得比较容易，使设计师和实现者之间有更好的交流，也可能构造出更好的工具去支持设计师和程序员。

例如，大部分设计方法都很关注程序的不同部分之间的依赖关系（通常是想使依赖关系最小化，并保证它们是定义良好的、可理解的）。如果一种语言能支持程序各部分之间的显式界面，那么它就能支持这一设计概念，也能保证实际存在的只有那些实际期望的依赖关系。由于在这种语言里的依赖关系都显式地写出来了，这样就可以提供读入程序产生依赖图的工具。这也使设计师和其他需要理解程序结构的人的工作更容易些。像C++这样的语言可以用于减小设计与程序之间的隔阂，从而降低出现混乱和误解的范围。

C++里最关键的观念是类。一个C++类就是一个类型。与名字空间一起，类也是一种最基本的信息隐藏机制。程序可以在用户定义类型和这些用户定义类型的层次结构的基础上描述。内部的和用户定义的类型都遵守静态类型检查规则。虚函数是一种运行时的约束机制，但又没有打破静态类型规则。模板支持参数化类型的设计。异常是一种使错误处理更加规范的方式。这些C++特征能够以不会引起超过C程序的额外开销的方式使用。这些都是C++里第一级的特性，设计师必须能理解和使用。除此之外，一般可用的重要的库（例如矩阵库、数据库界面、图形用户界面库、并行支持库）都会对设计的选择产生深刻影响。

对新事物的恐惧有时会导致对C++的并非最优化的使用，把从其他语言、系统和应用领域学到的东西错误地用到这里也会出现同样情况。设计师不能很好地利用语言特征以及不能重视有关的局限性的五种情况值得特别提出来：

- [1] 忽视类，所表述的设计方式将实现者限制到只能去使用C子集。
- [2] 忽视派生类和虚函数，只使用数据抽象子集。
- [3] 忽视静态类型检查，所表述的设计方式将实现者限制到只能去模拟动态类型检查。
- [4] 忽视程序设计，以一种排斥程序员的方式去描述系统。
- [5] 忽视除了类层次结构之外的其他所有东西。

出现这些情况的设计师通常是分别具有：

- [1] 某种C、传统CASE或者结构化设计的背景。
- [2] 某种Ada83、Visual Basic或者数据抽象的背景。
- [3] 某种Smalltalk或者Lisp的背景。
- [4] 某种非技术性的或者极其特殊的背景。
- [5] 某种极端强调“纯”面向对象程序设计的背景。

在每种情况下，都应该怀疑所用实现语言的选择是否正确，所用设计方法的选择是否正确，或者，是不是设计师本身不能适应手头的工具。

在这种不适应中，并没有什么不寻常的或者丢脸的。这也就是一种不适应，以至提交出的并不是最好的设计，给程序员增加了不必要的负担。当一种设计方法的概念框架明显地比C++的概念框架贫乏时，设计师也会出现同样的情况。因此，我们应该在所有可能的地方避免这种不适应性。

下面的讨论将分别回答这些异议，因为它们都经常在现实生活中出现。

24.2.1 忽视类

考虑忽视类的设计。结果做出的C++程序大致等价于经过同样设计过程做出的C程序，而这个程序又大致等价于经过同样设计过程做出的COBOL程序。在本质上，这一设计被做成“与程序设计语言无关的”，付出的代价就是迫使程序员在C和COBOL的公共子集里写代码。这一方法也确实有优点。例如，结果中数据和代码的严格分离使它比较容易使用传统的数据库，因为它们的设计就是针对这类程序的。因为使用的是最小化的程序设计语言，看起来对程序员的技能要求也比较低，或至少说对掌握多种技能的要求较低。对于许多应用（譬如说，传统的顺序性数据库更新程序），这种考虑问题的方式也是相当合理的，而且，使用多年的传统开发技术也适用于这种工作。

然而，假定这个应用与传统的顺序处理记录（或者字符）的方式有很大差异，或者所涉及的复杂性非常高，比如说，处于一个交互式的CASE系统中。由于设计中忽视了类，引起语言对数据抽象缺乏支持，就会造成实际的损害。内在复杂性将在应用中的某些地方显示出来。如果系统采用某种软弱无力的语言去实现，代码将无法直接反应设计，这样，程序将有太多代码行，缺乏类型检查，而且一般地说将难以使用工具。这也就预示着一个维护的噩梦。

对于这类问题的常用临时补救措施是做一些特殊工具来支持该设计方法的记述形式。用这些工具提供高层的结构和检查，弥补（有意造成的）实现语言能力低下的缺陷。这样，该设计方法就会变成一种专门用途的而且通常是公司所拥有的程序设计语言。在大部分环境中，与广泛可用而且具有适当的工具支撑的通用程序设计语言相比，这种程序设计语言都只能是拙劣的替代品。

在设计中忽视类的最常见原因就是惰性。传统程序设计语言不支持类的概念，传统的设计技术将反应出这种缺位，设计中最常见的关注点就是如何把问题分解为能执行所需动作的一组过程。这一概念在第2章里称为过程性程序设计，在设计的环境里被称做功能分解。一个时常听到的问题是：“我们能与一种基于功能分解的设计方法一起使用C++吗？”当然可以，但是你将很可能停止在将C++简单地用做一个更好的C，从而将遭遇前面所提到的各种问题。在转变的过程中，对于一个已经完成了的设计，在那些类机制看起来不能提供显著利益的子系统里（由于参与者们当时的经验所限），这样做都是可以接受的。但是，从长远的一般的观点看，由功能分解所蕴涵的反对大规模使用类的政策是与有效使用C++或者其他支持抽象的语言不相容的。

程序设计的面向过程和面向对象的观点之间存在着本质性的差异，对于同一个问题，它们通常会导向截然不同的解。这一观点对设计阶段也是正确的，就像它对实现阶段正确一样：你可以将设计集中于所采取的动作或者集中于所表示的实体，但无法同时做这两者。

那么，为什么要偏爱“面向对象的设计”，超过基于功能分解的传统设计方法呢？第一层的回答是，功能分解方法将导致不充分的数据抽象。正是由于这个因素，作为结果的设计将：

- 对于变化的弹性不够。
- 对于工具的顺应性不够。
- 对于并行开发不够合适。
- 对于并发执行不够合适。

问题在于，功能分解将使那些有意思的数据都成为全局性数据。因为当系统由一棵函数树构

成时,任何由两个函数访问的数据就必须对两者是全局性的。这样,访问某些“有意思的”数据的函数越多,就越要推动它浮向树的根部(作为计算领域的常规,树从根向下生长)。在单根性类层次结构中也可以看到完全相同的现象,其中“有意思的”数据和函数浮向根类(24.4节)。把注意力集中于类的描述和数据封装就是为了处理这个问题,使程序不同部分间的依赖关系明确化,更易驾驭。不过,最重要的是,通过数据引用的局部化,减少了系统里的相互依赖。

然而,有些问题的最好解决方式就是写出一组过程。“面向对象”设计途径的要点并不是说在程序里绝不能有非成员函数,或者系统中的任何部分都不能是面向过程的。与此相反,关键问题在于松弛程序不同部分之间的联系,以更好地反映应用中的概念。在典型情况下,将设计工作的基本点放在类上而不是函数上,能最好地达到这个目的。过程风格的使用应该是有意识的决策,而不应该是默认方式。类和过程都应相对于应用而适当地使用,不应该成为某种僵硬设计方法导致的人为现象。

24.2.2 忽视继承

现在考虑忽视继承的设计。结果程序将不能利用一种C++关键特征的优势,但仍然会得到C++中许多C、Pascal、Fortran、COBOL等所没有的优点。这样做的常见原因(除了惰性之外)是声称“继承是一种实现细节”、“继承破坏了信息隐藏”以及“继承使得与其他软件的合作更加困难”。

认为继承仅仅是一种实现细节,就是忽略了类层次结构的方式可以直接模拟应用领域中概念之间的关系。这种关系在设计中应该是明显的,以使设计师能对它们进行推理。

可以设想另一种更强的情况:将继承排斥于某个C++程序的各个部分之外,是因为该程序必须直接与用其他语言写出的程序联系。然而,这并不是在整个系统里避免使用继承的理由,它不过说明需要封装起程序,仔细描述好与“外部世界”的界面。类似地,有关继承的使用可能损害信息隐藏的担心(24.3.2.1节)只是说明,应该细心使用虚函数和保护成员(15.3节)。这些都不应该成为普遍地排斥继承的理由。

在许多情况下,从继承不会获得任何实际利益。然而,在大型项目里,“不用继承”的政策的结果将是不易理解的不够灵活的系统,其中还会用更传统的语言和设计结构去“伪装”继承。进一步说,我想,即便有此政策,继承最终还是会被到处使用,因为C++程序员将会在系统里各个部分中,为基于继承的设计找到许多有说服力的证据。这样,“不用继承”的政策将必然导致丧失一致性的整体结构,并将类层次结构的使用限制在特定子系统里。

换句话说,要保持一种开放的心态。类层次结构并不是每个好程序中必不可少的一部分。但是,在许多情况下,它们能有助于理解应用问题,有助于解的描述。继承可能误用,也可能被过度使用,但这些事实只能作为谨慎行事的理由,而不是禁止使用的理由。

24.2.3 忽视静态类型检查

现在考虑忽视静态类型检查的设计。在设计阶段忽视静态类型检查,最常听到的原因是:“类型是程序设计语言中的一种人为现象”,“考虑对象而不为类型操心更自然一些”,以及“静态类型检查迫使我们过早去考虑实现方面的问题”。这种态度是很好的,只要它不超过限度就无大碍。在设计阶段忽略类型检查的细节也很合理,在分析阶段和早期设计阶段,几乎

完全忽视类型问题通常也是安全的。但是，类和类层次结构在设计中很有用，特别是它们使我们能描述概念，能将概念间的关系精确化，能有助于我们做有关概念的推理。随着设计的进展，这种精确性的反应形式就是有关类及其界面的越来越精确的描述。

最重要的是应认识到，精确描述和强类型界面是一种基本设计工具。C++语言的设计一直把这些牢记在心。一个强类型的界面能保证（直至某种程度）软件中只有相互兼容的片段才能通过编译并连接到一起，这就使这些软件片段可以对其他片段做出比较强的假设。这些假设由类型系统保证，其作用是尽可能减少了运行时检测的使用，从而能提高效率，也能显著缩短多人参与的项目的集成阶段。在集成那些提供了强类型界面的系统方面，人们取得了非常有力的正面经验。实际上，也正是因为这个原因，集成问题才没有被当做本章的主要论题。

再来考虑一个类比的例子。在物理世界中，我们时时需要将物品插接到一起，看来存在着无穷多种相互插接的标准。有关这些插接的最明显现象就是它们具有特殊的设计，以使除非两个物品的设计就是为了插到一起，否则就不能插到一起，而且它们只能按照正确的方式插接。你不能将一个电动剃须刀插入一个高压插座。如果你真能这样做，那结果就可能是一个烧毁的剃须刀和一个烧焦的剃须者。人们把许多别出心裁的设计用在保证不相容的硬件不能插接到一起的问题上。代替不相容插接件的另一种方式，就是让插接物在被插入插座时，能保护自己免于不当行为的损害。涌流保护器是这方面的一个很好实例。因为完全的相容性无法在“插接相容层次”上得到保证，我们有时会需要更昂贵的保护电路，以便动态适应输入的变化并/或保护自己。

这种类比几乎是精确的。静态类型检查等价于插接相容性，而动态检查对应于保护/适配电路。如果两种检查都失败了，无论是在物理世界，还是在软件世界，结果必定是严重的损害。在大型系统里同时使用着两种检查。在设计的最初阶段，简单地说“它们两者应当插接到一起”是合理。不过，它们应怎样插在一起的问题很快就会变得需要考虑了。这种插接应当提供什么行为？可能出现什么错误条件？对第一级代价的估计如何？

“静态类型”的应用不限于物理世界。利用各种单位（例如，米、公斤、秒等）防止不相容物件的混用，这类情况在物理和工程中随处可见。

在23.4.3节有关设计步骤的描述中，类型信息在第2步就进入我们的视野（假定在第1步粗略地考虑之后），到第4步就变成了主要问题。

静态类型检查是保证由不同小组开发的C++软件之间能相互合作的基本媒介。有关界面的文档（包括所涉及的准确类型）是分开工作的程序员小组间交流的基本手段。这些界面是设计阶段的输出中最重要的部分之一，也是设计师和程序员之间交流的汇合点。

如果在考虑界面时忽视类型问题，得到的设计将具有比较模糊的程序结构，并把错误检查推迟到运行时。举个例子，一个界面可以通过自标识对象来描述：

// 示例，假定动态类型检查，而非静态类型检查：

Stack s; // 堆栈里可保存任意类型对象的指针

void f()

{

s.push(new Saab900);

s.push(new Saab37B);

s.pop()->takeoff(); // 可以：Saab 37B是飞机

s.pop()->takeoff(); // 运行时错误：汽车不能起飞

}

这是一个严重的规范不足的界面 (*Stack::push()*), 其着眼点在于动态检查而不是静态检查。堆栈*s*的本意是保存*Plane*, 但在代码中对这件事却秘而不宣, 因此就使保证这个要求得到满足变成了用户的责任。

另一种更精确的规范, 即通过模板加虚函数而不是无约束的动态类型检查, 可以把对错误的检查从运行时期提前到编译时期:

```
Stack<Plane*> s; // 存放指向Plane指针的堆栈

void f()
{
    s.push(new Saab900); // 错误: Saab900不是飞机
    s.push(new Saab37B);

    s.pop()->takeoff(); // 可以: Saab 37B是飞机
    s.pop()->takeoff();
}
```

在16.2.2节提出了类似的论点。动态检查和静态检查在运行时间上的差异可能非常明显, 动态检查的额外开销通常在3到10倍的样子。

当然, 也不应该走到另一个极端。静态检查不可能捕捉到所有的错误。例如, 一个程序即使经过了最彻底的静态检查, 对于硬件失误也无能为力。另见25.4.1节, 那里有一个不可能进行完全的静态检查的例子。然而, 理想还是使最大部分的界面通过应用层次的类型完成静态检查, 参见24.4.2节。

另一个问题是, 一个在抽象层次上完全合理的设计, 因为没有将所用的基本工具 (这里是C++) 的限制考虑在内, 也可能导致严重的麻烦。举例说, 函数*f()*需要对某参数执行操作*turn_right()*, 但这样做的条件是它的所有参数有一个公共类型:

```
class Plane {
    // ...
    void turn_right();
};

class Car {
    // ...
    void turn_right();
};

void f(X* p) // X的类型应该是什么?
{
    p->turn_right();
    // ...
}
```

在有些语言 (例如Smalltalk或者CLOS) 里, 只要两个类型有同样的操作, 就允许它们的互换使用, 其中通过一个公共基类联系起各个类型, 对名字解析被推迟到运行时进行。然而, C++ (有意) 通过模板和编译时检查来支持这一概念。一个非模板函数能接受两个类型的参数, 条件是这两个类型能隐式地转换到某个公共类型。这样, 上例中的*X*就必须是*Plane*和*Car*的一个公共基类 (例如一个*Vehicle*类)。

典型情况是, 那些受到与C++无关的概念启发而产生的例子, 可以通过将有关的假设明确化而映射到C++里。例如, 有了*Plane*和*Car* (没有公共基类), 我们还是可以创建起一个类层次结构, 以使我们能够传递一个包含*Plane*或者*Car*的对象给*f(X*)* (25.4.1节)。当然, 做这种

事情通常需要太多的机制和智慧。模板常常是完成这种概念映射的有用工具。设计概念与C++之间的不匹配通常会导致“看起来很不自然”的低效代码。做维护的程序员特别不喜欢由于这种不匹配而产生的非习见形式的代码。

在设计技术与实现语言间的不匹配相当于自然语言间的逐字翻译。例如，具有德语语法的英语和具有英语语法的德语一样丑陋，对于某些只熟悉两种语言之一的人而言，它们几乎是不可理解的。

程序里的类是设计中的概念的具体体现。因此，使类之间的关系难以理解，也就是使设计中的基本概念难以理解。

24.2.4 忽视程序设计

与许多其他活动相比，程序设计是代价高昂且无法预计的，结果代码也常常不能100%可靠。程序设计是劳动密集型的，而且，由于各种原因，大部分认真的项目都明显地脱期，由于代码没有做好而无法发布。所以，为什么不将程序设计这种活动彻底清除呢？

对于许多管理者来说，摆脱这些妄自尊大、缺乏修养、多拿报酬、鬼迷心窍、穿着不合礼仪的程序员^①看起来是大有好处的事。对程序员而言，这些听起来就十分荒谬了。当然，也确实有一些重要问题领域，在那里存在着替代传统程序设计的方式。对于特定的领域，确实有可能从高层次的规范直接生成代码。在另一些领域里，可以通过直接在屏幕上操作而得到代码。例如，通过直接操作构造出有用的用户界面所花费的时间，比通过传统程序设计构造出同样界面要少得多。类似地，数据库布局和依据这种布局访问数据的代码可以通过规范生成，这样做，比直接在C++或者其他任何通用程序设计语言里表述这些代码简单得多。通过规范生成或者通过直接的界面操纵产生出的状态机，通常比大部分程序员写出的更小、更快且更正确。

这些技术在特殊领域中工作得非常好，只要在这里存在可靠的理论基础（例如，数学、状态机、关系数据库等）或者存在着某种通用框架，使得小的应用片段能够嵌入其中（例如，图形用户界面、网络模拟、数据库模式等）。这些技术在有限的（常常也是意义最大的）领域中表现出明显的价值。这种情况就诱使人们去设想，通过这些技术驱逐传统程序设计是“指日可待”的了。不对！原因是，将规范技术扩展到存在有效理论框架之外的领域，也就意味着规范语言本身将需要有传统程序设计语言的复杂性。这一点就足以摧毁创建一个清晰的基础良好的规范语言的目标。

有时人们忘记了如下事实：那些能在一个领域中摒弃传统程序设计的框架，本身就是一个通过传统程序设计方式设计、编程、测试的系统或者一个库。事实上，C++以及本书中所描述的技术的一种流行应用就是设计和构造这些系统。

做出一种折衷，只提供通用程序设计语言的表达能力的一部分，在将这种东西应用到受限的应用领域之外时，对于两个世界都将是最坏的事情。集中关注高层模型观点的设计师将由于所增加的复杂性而感到极度烦恼，做出的规范将生成可怕代码。使用传统程序设计技术的程序员也遇到挫折，由于在这里缺乏语言的支持，只能通过超常努力或抛弃高层模型才能生成更好的代码。

① 是这样的，我就是个程序员。

我看不到有任何迹象能说明：除了在那些已经有了良好理论基础或通过框架提供了基本程序设计的领域中，程序设计作为一种活动可能被成功地清除。在这两种情况下，只要人离开了原来的框架并试图获得更通用的成果时，有关技术的有效性就会急剧下降。相反，在那些高层规范技术和直接操纵技术已经有了良好基础并相当成熟的领域里，忽视它们就更是一种愚蠢行为。

设计工具、库和框架是一种最高形式的设计和编程。为某个应用领域构造出一种有用的基于数学的模型是一种最高形式的分析。这样提供出的工具、语言和框架等将使这种工作的成果能为成千上万的程序员使用，这也是使设计师和程序员逃出陷阱和避免被变成制造某类人工制品的工匠的一种方法。

最重要的是，一个规范系统或者基础库应能与通用的程序设计语言有效接口。否则，所提供的框架就是从本质上受限的。这也意味着，如果规范系统和直接操纵系统能在某个适当的高层次产生出可接受的通用程序设计语言代码，这样的系统就会有很大的优势。专有的语言只能为其提供商带来长期利益。如果所生成的代码层次太低，加人的通用代码就只能在无法借助于抽象的情况下写出，这样就会失去可靠性、可维护性和经济性。从本质上看，生成系统的设计应当能结合起高层次规范和高级程序设计语言两者的威力。排斥其中的哪一个都必定会为工具制造者的利益而牺牲系统构造者或者工具构造者的利益。成功的大系统都是多层次的、模块化的、随着时间演化的。因此，生产这类系统的成功努力涉及到各种各样的语言、库、工具和技术。

24.2.5 排他性地使用类层次结构

当我们发现了某些新东西能实际工作时，常常会做些鲁莽事，会不分青红皂白地去应用它。换句话说，对某些问题的一个伟大的解决方案经常被看做是几乎所有问题的惟一正确的解决方案。类层次结构和对它们的（一个）对象的多态操作是处理许多问题的伟大解决方案。然而，并非每个概念最好就是表示为某个类层次结构的一部分，而且也并非某个软件部件最好就是表示为一个类层次结构。

为什么呢？一个类层次结构表述了它的类之间的关系，而一个类代表了一个概念。那么，什么是一张笑脸、我的CD-ROM的驱动程序、Richard Strauss的Don Juan录音、一行文字、一颗人造卫星、我的医疗记录和一个实时时钟之间的共同关系呢？当它们仅有的共享性质就是它们都是程序设计的制品（它们都是“对象”）时，将它们放入一个单一类层次结构将没有任何本质性的价值，而只能造成混乱（15.4.5节）。硬把所有东西塞进一个类层次结构里，将会引进一些人为的相似性，并搞模糊真实的关系。只有通过分析揭示出的概念间的共性，或是在设计和编程中发现了结构中存在有利于实现概念的有用共性时，才应该使用类层次结构。对于后一种情况，我们还需要仔细区分真正的共性（应该通过公用继承反应为子类型关系）或用于简化实现（反应为私用继承，24.3.2.1节）。

这种思考线路带给程序的将是若干相互无关的或者弱相关的类层次结构，每个都表示了一组紧密相关的概念。它也会导致具体类的概念（25.2节），这些类不在任何类层次结构里，因为将其放入可能损害性能或者损害与系统其他部分之间的独立性。

为了产生效用，在作为类层次结构中一个组成部分的类里，大多数关键性操作都必须是虚函数。进一步说，这种类的大部分数据必须是保护的而不是私用的，而这也将使它容易受到来自派生类的修改的损害，这也将大大增加测试的复杂性。从设计的观点看，如果在某些

地方做更严格封装确实有意义,那么就应该采用非虚函数或者私用数据(24.3.2.1节)。

让操作的某一个参数特殊(它用于指定“那个对象”)也有可能产生扭曲的设计。当几个参数最好是同样对待时,该操作最好就是表示为非成员函数。这并不意味着该函数就是全局的,事实上,几乎所有这种自立的函数都应该是某个名字空间的成员(24.4节)。

24.3 类

面向对象的设计和编程中最基本的观念就是:程序应该是现实中某些方面的模型。程序里的类代表着应用中的基本概念,特别是被模型化的“现实”的基本概念。真实世界的对象和实现中所需的实体都通过这些类的对象来表示。

有关类之间关系和类中各个部分间关系的分析是系统设计的中心:

24.3.2节 继承关系。

24.3.3节 包容关系。

24.3.5节 使用关系。

24.3.6节 编在程序里的关系。

24.3.7节 类内的关系。

因为C++的类就是类型,类和类之间的关系从编译器那里得到了强有力的支持,一般而言都服从于静态检查。

要在一个设计中起作用,一个类不但要表示某个有用的概念,而且还必须提供一个适当的界面。简而言之,类应该对其外部世界有一个极小化且定义良好的依赖关系,它给出了一个界面,向世界的其余部分提供了必须而又是最小量的信息(24.4.2节)。

24.3.1 类表示什么

本质上说,在系统里存在着两种类:

[1] 直接反映应用领域中概念的类,这里的概念指的是那些被最终用户用于表述其问题或者解的概念。

[2] 为了实现而人工创造的类,也就是设计师和程序员用于描述他们的实现技术的那些类。有些类是为了实现而人工创造的,但又代表着真实世界中的某些实体。举例说,系统中的软件和硬件资源也是应用中类的很好候选者。这反映了一个事实,一个系统可以从多种角度去观察。这也意味着,一个人的实现细节可能就是另一个人的应用。良好设计的系统应该包含着一些类,用于支持系统的不同逻辑视图。例如:

[1] 表示用户层概念的类(如汽车和卡车)。

[2] 表示用户层概念的推广的类(如交通工具)。

[3] 表示硬件资源的类(如一个存储管理类)。

[4] 表示系统资源的类(如输出流)。

[5] 用于实现其他类的类(如表、队列和锁)。

[6] 内部类型和控制结构。

在那些比较大的系统里,使逻辑上属于不同种类的类保持分离,并维持不同抽象层次的分离,是一项具有挑战性的工作。一个简单例子可以被认为具有三个抽象层次:

[1+2] 提供该系统的一个应用层视图。

[3+4] 表示这个模型运转于其上的机器。

[5+6] 代表有关实现的一个低层的（程序设计语言的）视图。

系统越大，为描述该系统所需要的抽象层次通常也就越多，定义和维护这些层次也就越困难。注意，这种抽象层次在自然界或者其他类型的人工制品方面有着直接的对应物。例如，一座房子可以认为是由下面各种事物组成的：

[1] 原子。

[2] 分子。

[3] 木材和砖头。

[4] 地面、墙和天花板。

[5] 房间。

只要保持了这些抽象层次的相互分离，你就可以维持对于房子的一种具有内在一致性的视图。当然，如果你混淆了它们，那就会出现荒谬的情况。举例来说，陈述“我的房子由数千斤碳，一些复杂的聚合物，大约5000砖头，两个浴室和13个天花板组成”完全是蠢话。由于软件的抽象本性，有关复杂系统的类似陈述却未必总能被看清楚。

将应用领域中的概念翻译到设计中的类绝不是一个简单的机械性操作，通常都需要相当的洞察力。注意，应用领域里的概念本身常常也是抽象，例如，“纳税人”，“僧侣”和“雇员”在现实中都不存在，这些概念本身就是一种加在个人身上的标签，是为了在某些系统里对人们进行分类。真实的甚至是虚幻的世界（文学，特别是科幻）有时也成为有关概念的想法之源，在转化为类的时候常常剧烈地变异。举例来说，我的PC屏幕并不真正像我的桌面，尽管其设计支持一种桌面的感觉^①，而我屏幕上的窗口与那些允许微风吹进我办公室的新鲜设计之间只有极少的关系。关于模拟真实世界的论点并不意味着要奴隶式地追随我们之所见，而是要以它们作为设计的出发点、灵感的源泉。用它们作为一种铁锚，当软件难以对付的本性威胁着要战胜我们理解自己程序的能力时，用于去抓住它。

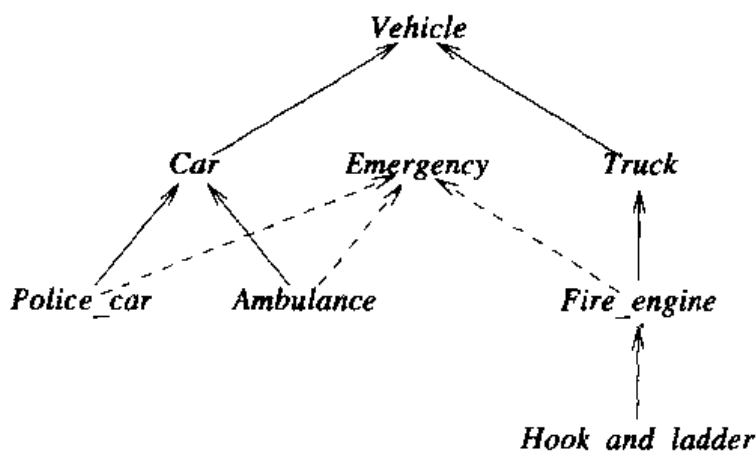
一点告诫：初学者常常觉得很难“发现类”，但这个问题不久就会解决，并没有长期的严重影响。但随之而来的是这样一个阶段，其中类（及其相互关系）看起来出现了无法控制的膨胀。这样就会造成结果程序在复杂性、可理解性和效率方面的长期性问题。并不是每个小小的细节都应该用一个单独的类表示，也不是类间的所有关系都需要表达为类继承。应该试着记住，设计的目标是在某个适当的细节层次上和在某些适当的抽象层次上去模拟一个系统。在简单性与普遍性间找到一种平衡可不是件容易的事情。

24.3.2 类层次结构

现在考虑模拟一个城市里的交通流，以确定急务车辆大约需要多长时间才能达到目的地。很清楚，我们需要表示小汽车、卡车、救护车、不同种类的救火车、警车、公共汽车等。继承将会参与进来，因为真实世界里的概念不是孤立存在的，而是与其他概念有着各种关系。没有对这些关系的理解，我们就不可能理解这些概念。因此，如果一个模型中没有表现出这些关系，它也就不可能恰当地表示有关的概念。也就是说，在我们的程序里需要用类去表示概念，但这还不够，我们还需要采用某些方式去表示类之间的关系。继承是一种直接表示层

^① 无论如何，我可能无法忍受在我的屏幕上出现那样的混乱状态。

次关系的强有力手段。在当前的这个例子里，我们可能需要对急务车辆做一些特殊处理，也希望能区分小汽车一类的车辆和卡车一类的车辆。这样就会沿着这些线产生一个类层次结构：



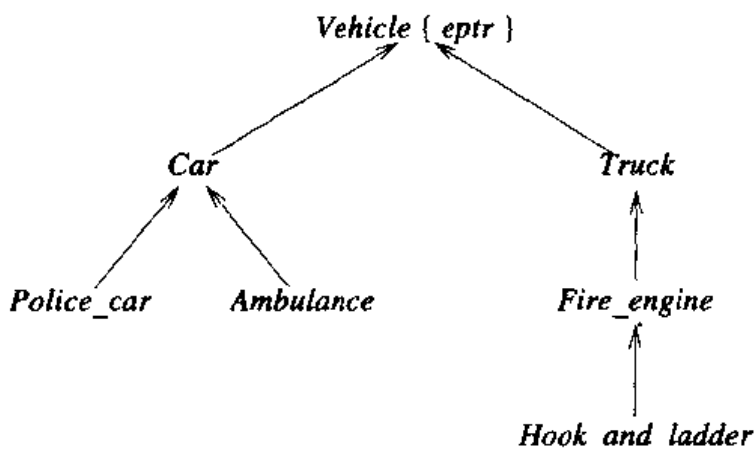
在这里，**Emergency**表示的是急务车辆的一些与这个模拟相关的侧面：它可以违反某些交通规则，在为响应应急电话而会车时具有优先权，在派遣员的控制之下，等等。

下面是C++里的定义：

```

class Vehicle { /* ... */ };
class Emergency { /* ... */ };
class Car : public Vehicle { /* ... */ };
class Truck : public Vehicle { /* ... */ };
class Police_car : public Car, protected Emergency { /* ... */ };
class Ambulance : public Car, protected Emergency { /* ... */ };
class Fire_engine : public Truck, protected Emergency { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };
  
```

继承是一种可以在C++里直接表示的高层关系，也是在设计的早期阶段就能大范围出现的关系。在这里常常有选择，是用继承表示某种关系还是通过成员。也可以考虑有关什么是急务车辆的另一种观念：凡是能显示某种闪烁的标志灯的车辆就是急务车辆。这样做将能简化上述类层次结构，用类**Vehicle**的一个成员代替**Emergency**类：



现在，类**Emergency**被简单地用做类里的一个成员，在需要作为急务车辆时使用：

```

class Emergency { /* ... */ };
class Vehicle { protected: Emergency* eptr; /* ... */ }; // 更好的：提供到eptr的完整界面
class Car : public Vehicle { /* ... */ };
  
```

```

class Truck : public Vehicle { /* ... */ };
class Police_car : public Car { /* ... */ };
class Ambulance : public Car { /* ... */ };
class Fire_engine : public Truck { /* ... */ };
class Hook_and_ladder : public Fire_engine { /* ... */ };

```

在这里，一辆车是急务车辆，如果它的`Vehicle::eptr`不是零，“普通的”小汽车和卡车都将`Vehicle::eptr`初始化为0，其他则将`Vehicle::eptr`初始化为非零。例如，

```

Car::Car() // Car的构造函数
{
    eptr = 0;
}

Police_car::Police_car() // Police_car的构造函数
{
    eptr = new Emergency;
}

```

按照这种方式定义事物，将能允许从急务车辆到普通车辆的转换或者反过来：

```

void f(Vehicle* p)
{
    delete p->eptr;
    p->eptr = 0; // 不再作为急务车辆

    // ...

    p->eptr = new Emergency; // 重新作为急务车辆
}

```

那么，哪一种类层次结构形式最好呢？一般性的回答是：“能够直接模拟真实世界中我们所感兴趣的那些方面的程序是最好的。”也就是说，在选择模型时，我们应该在不可避免的效率 and 简单性的约束下，将目标定在尽可能现实上。在当前情况下，在常规车辆与急务车辆之间的简单转换在我看来并不现实。救火车和救护车都是配备了经过特殊训练人员的专用车辆，通过特殊派遣程序投入操作，需要特别的通信设备。这一观点说明，急务车辆应当作为程序中的一个基本概念直接表示，以利于类型检查和其他工具的使用。假设我们是在另一个地方做模拟，其中车辆的角色较少固定（比如说在某个地方，那里私用汽车被程序性地用于装运急务人员到事故地点，通信全部基于普通移动电台），那么，采用另一种模拟系统的方式可能就更合适些。

对那些认为交通模拟问题难以理解的人，在这里值得指出，这种在继承和成员关系之间权衡的情况几乎不可避免地会在设计中出现。24.3.3节中的滚动条示例是另一个等价的例子。

24.3.2.1 类层次结构内的依赖性

很自然，派生类将依赖于它的基类。认识到反过来也成立的人就少多了^①。如果某个类有一个虚函数，只要它的派生类里覆盖了这个函数，这个类就将依赖于其派生类去实现它的部分功能。如果某个基类的一个成员调用了该类里的一个虚函数，那么这个类就在自己的实现中依赖于派生类了。与此类似，如果某个类里使用到保护成员，它自己的实现也就依赖于它的派生类了。考虑：

^① 这个观点可以总结为“精神错乱是遗传的，你从你的孩子那里得到它。”

```
class B {
    // ...
protected:
    int a;
public:
    virtual int f();
    int g() { int x = f(); return x-a; }
};
```

函数`g()`到底做什么？答案是，由某个派生类中`f()`的定义决定。下面这个版本将保证`g()`对`a`加1并返回1：

```
class D1 : public B {
    int f() { return a+1; }
};
```

下面是一个使`g()`写出“*Hello world!*”并返回0的版本：

```
class D2 : public B {
    int f() { cout<<"Hello, world!\n"; return a; }
};
```

这个例子展示了有关虚函数的一个最重要的情况。为什么这么傻？为什么程序员还要写出像这样的东西？答案是，虚函数是基类界面的一部分，而且假定在不必知道由它派生的类的情况下就可以使用这样的类。为此，就必须能描述这一基类的对象的预期行为，使得可以在没有任何有关派生类的知识的情况下写出程序来。例如，类`Shape`的虚函数`rotate()`完成形状的旋转，在像`Circle`或`Triangle`等派生类里的`rotate()`就必须去旋转它们各自类型里的对象，否则，就破坏了有关`Shape`类的基本假设。对于上面的类`B`与其派生类`D1`和`D2`的行为方面没有任何假设，因此这个例子就是一派胡言。包括`B`、`D1`、`D2`、`f`和`g`的名字选择也是为了不夹带任何可能的意义。对于虚函数的行为期望的规范也是类设计时需要重点关注的一个问题。为类和函数选择好的名字非常重要——但却常常很不容易。

依赖于未知（或许仍然继续未知）的派生类是好还是坏？自然，这种依赖性是基于程序员的意图。如果本意在于将一个类与所有的外部影响隔绝开，使它具有某种可证明的特定行为方式，那么最好是避免保护成员和虚函数。然而，如果本意就是提供一个框架，使以后的程序员（例如几周后的同一个程序员）可以加入代码，那么虚函数就是达到这种效果的最美妙机制，而保护成员函数已被证明对于支持这类使用是很方便的。这一技术被用在流I/O库（21.6节），也在`Ival_box`层次结构的最后版本里展示（12.4.2节）。

如果某个`virtual`函数只是想由某个派生类间接地使用，那么也可以让它是`private`。例如，考虑下面这个简单的缓冲区模板：

```
template<class T> class Buffer {
public:
    void put(T);    // 如果缓冲区满则调用overflow(T)
    T get();       // 如果缓冲区空则调用underflow()
    // ...
private:
    virtual int overflow(T);
    virtual int underflow();
    // ...
};
```

函数`put()`和`get()`将分别调用`overflow()`和`underflow()`。现在用户就可以通过覆盖`overflow()`和`underflow()`来实现满足各种需要的缓冲区类型了:

```
template<class T> class Circular_buffer : public Buffer<T> {
    int overflow(T);    // 如果满就循环使用
    int underflow();
    // ...
};

template<class T> class Expanding_buffer : public Buffer<T> {
    int overflow(T);    // 如果满就增大缓冲区
    int underflow();
    // ...
};
```

只有在派生类需要直接调用`overflow()`和`underflow()`时,才应该将它们作为`protected`而不是`private`。

24.3.3 包容关系

在使用包容时,对于表示类`X`的对象,存在着两种主要的可选方式:

- [1] 声明一个类型为`X`的成员。
- [2] 声明一个类型为`X*`或者`X&`的成员。

如果指针值绝不改变,那么,除了效率问题和你写构造函数、析构函数的方式之外,这些选择方案是完全等价的:

```
class X {
public:
    X(int);
    // ...
};

class C {
    X a;
    X* p;
    X& r;
public:
    C(int i, int j, int k) : a(i), p(new X(j)), r(*new X(k)) { }
    ~C() { delete p; delete &r; }
};
```

在这些情况下,通常应该优先采用对象本身的成员关系(这里是`C::a`),因为它在时间、空间和击键次数上都是效率最高的。这样做也不容易出错,因为被包容对象与包容对象的联系已经包含在构造和析构关系里面了(10.4.1节、12.2.2节、14.4.1节)。不过还请看看24.4.2节和25.7节。

如果在“包容”对象的生存期间需要改变“被包容”对象,那么就应该采用指针解决方案。例如,

```
class C2 {
    X* p;
public:
    C2(int i) : p(new X(i)) { }
    ~C2() { delete p; }

    X* change(X* q)
    {
```

```

        X* t = p;
        p = q;
        return t;
    }
};

```

采用指针成员的另一原因是允许通过参数提供“被包容”对象：

```

class C3 {
    X* p;
public:
    C3(X* q) : p(q) { }
    // ...
};

```

通过让对象包含一个指向其他对象的指针，我们就建立起了通常所说的对象层次结构。这可以作为类层次结构的一种替代方式或者补充方式。如24.3.2节急务车辆的例子所示，在选择将某类作为基类还是作为成员之间也存在一个不易处理的选择问题。如果需要覆盖，那就表明前一种方式是更好的选择。相反，如果需要将某种性质用一些类型表示，则表明后者是更好的选择。例如，

```

class XX : public X { /* ... */ };
class XXX : public X { /* ... */ };
void f()
{
    C3* p1 = new C3(new X);    // C3 “包含” 一个X
    C3* p2 = new C3(new XX);   // C3 “包含” 一个XX
    C3* p3 = new C3(new XXX);  // C3 “包含” 一个XXX
    // ...
}

```

让C3从X派生或让X有一个C3成员的方式都不能模拟这种情况，因为成员也需要使用准确的类型。这一点对于有虚函数的类非常重要，例如形状类（2.6.2节）或抽象集合类（25.3节）。

如果在包容对象的生存期间被引用的一直是同一个对象，那么就可以用引用来简化基于指针成员关系的类。例如，

```

class C4 {
    X& r;
public:
    C4(X& q) : r(q) { }
    // ...
};

```

如果某对象需要共享，那么也必须用指针或者引用成员：

```

X* p = new XX;
C4 obj1(*p);
C4 obj2(*p); // obj1和obj2现在共享这个新XX

```

很自然，共享对象的管理需要格外小心，特别是在并发系统里。

24.3.4 包容和继承

由于继承关系的重要性，它时常被错误使用、错误理解也就不奇怪了。当类D是以公用方

式从另一个类**B**派生出时，人们经常说一个**D**就是一个**B**：

```
class B { /* ... */ };
class D : public B { /* ... */ };    // D是一种B
```

换种说法，这也可以表述为说继承是“是一个”关系，或者（或许更准确些）**D**是一种**B**。与之相对应的是，类**D**里有一个成员属于另一个类**B**，则通常被说成是有一个**B**或者包含一个**B**。例如，

```
class D { // 一个D里包含一个B
public:
    B b;
    // ...
};
```

换种说法，这也可以表述为，成员关系是“有一个”关系。

对于确定的**B**和**D**，我们应如何在继承和成员关系之间做出选择呢？考虑**Airplane**和**Engine**。新手常常会去琢磨，让**Airplane**作为**Engine**的派生类是不是一个好主意。这当然是个坏主意，因为一个**Airplane**不是一个**Engine**；它有一个**Engine**。看这件事的一种方式考虑一个**Airplane**是否可以有两个或者更多的**Engine**。因为这看来很合理（即使是在我们所考虑的程序里，所有的**Airplane**都是单发动机的），所以我们应该用成员关系而不是继承。在有疑问之处，问题“它能不能有两个？”在许多地方都很有用。与通常一样，软件不可琢磨的性质使这个讨论很有意义，如果所有的类都像**Airplane**和**Engine**这样容易看见，像让**Airplane**从**Engine**派生这一类平凡错误就太容易避免了。然而，这类错误却经常能看到，特别是在那些把派生简单地看做程序设计语言层结构中的另一种组合机制的人员那里。尽管派生的使用很方便，也提供了记法上的简写形式，但基本上只应该将它用于表述设计中定义良好的关系。考虑

```
class B {
public:
    virtual void f();
    void g();
};

class D1 {          // 一个D1包含一个B
public:
    B b;
    void f();        // 不覆盖b.f()
};

void h1(D1* pd)
{
    B* pb = pd;      // 错误：没有D1* 到B* 的转换
    pb = &pd->b;
    pb->g();           // 调用B::g()
    pd->g();           // 错误：D1没有成员g()
    pd->b.g();
    pb->f();           // 调用B::f( 没被D1::f()覆盖)
    pd->f();           // 调用D1::f()
}
```

注意，并不存在从一个类到它的成员的隐式转换。当一个类里包含一个其他类的成员时，它也不能去覆盖那个成员的虚函数。这与公用派生的情况不同：

```

class D2 : public B {    // 一个D2是一个B
public:
    void f();           // 不覆盖B::f()
};

void h2(D2* pd)
{
    B* pb = pd;         // 可以：隐式D2* 到B* 的转换
    pb->g();             // 调用B::g()
    pd->g();             // 调用B::g()
    pb->f();             // 虚调用：调用D2::f()
    pd->f();             // 调用D2::f()
}

```

由D2例子提供的写法比D1例子更方便，这也是导致过度使用派生的一个因素。但应该记住，在取得这些写法上方便的同时，也付出了增加B与D2间依赖关系的代价（见24.3.2.1节）。特别是容易忘记从D2到B的隐式转换。除非你的类的语义方面能接受这种转换，否则就应避免采用公用派生。当某个类被用于表示一个概念而派生正是表示了是一个关系时，这种转换通常就是我们所希望的。

也存在一些情况，在其中你需要派生，但又不能接受这种转换的发生。考虑写一个Cfield（controlled field，控制域）类，它除了做自己该做的其他事情外，还提供对另一个类Field的运行访问控制。乍一看，通过从Field派生来定义Cfield似乎很正确：

```
class Cfield : public Field { /* ... */ };
```

这正是表示了Cfield是一种Field的观念，在写Cfield函数时允许采用方便的写法，直接去使用作为Cfield的一部分的Field成员，而且最重要的是允许Cfield去覆盖Field的虚函数。暗礁是从Cfield* 到Field* 的隐式转换，这也意味着，Cfield的声明方式摧毁了所有想控制对Field的访问的企图：

```

void g(Cfield* p)
{
    *p = "asdf";        // 通过Cfield的赋值运算符访问被控的Field
                        // p->Cfield::operator = ("asdf")

    Field* q = p;        // 隐式的Cfield* 到Field*转换
    *q = "asdf";         // 呜呼！没法控制
}

```

一种解决办法就是让Cfield有一个Field成员，但这样做就完全排除了Cfield覆盖Field的虚函数的可能性。更好的解决办法是采用私用派生：

```
class Cfield : private Field { /* ... */ };
```

从设计的观点看，私用继承等价于包容，除了（偶尔必须的）覆盖问题之外。这种方式的一个重要应用是一种技术，这种技术让一个类从一个定义界面的抽象基类通过公用方式派生，同时让它从另一个提供实现的具体类通过私用或者保护方式派生（2.5.4节、12.3节、25.3节）。由于私用或保护派生所隐含的继承是一种并不反应在派生类的类型中的实现细节，这种继承有时被称为实现继承。与此相对的是公用派生，不仅基类的界面被继承，也允许到基类型的隐式转换。后一种继承有时也被称为子类型或者界面继承。

陈述这一情况的另一种方式是指出：派生类的对象应该能够用到其基类对象能够使用的所有地方。这有时被称为“Liskov替换原理”（23.6节 [Liskov, 1987]）。公用/保护/私用的划分

能直接支持这种概念，以通过指针或者引用去操作多态类型。

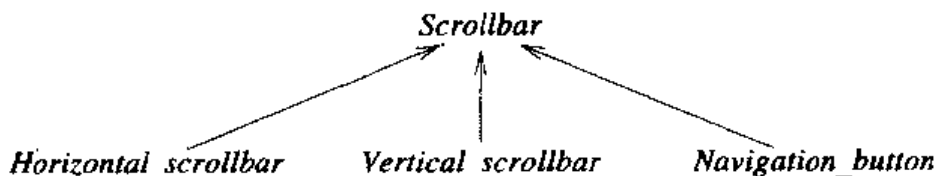
24.3.4.1 成员/层次结构间的权衡

为进一步考察涉及包容和继承的设计选择，现在考虑如何在一个交互式的图形系统里表示一个滚动条，以及如何将滚动条附着到窗口上。我们需要两类滚动条：水平的和垂直的。我们可以通过两个类型 *Horizontal_scrollbar* 和 *Vertical_scrollbar* 表示它们，或者用一个类型 *scrollbar*，其中用一个参数指明其布局是水平的还是垂直的。前一种选择意味着需要三个类型，要用一个普通 *Scrollbar* 类型作为两个特殊滚动条类型的基类。后一种选择意味着滚动条类型里需要一个额外的参数，还需要为表示滚动条的种类选择一些值。例如，

```
enum Orientation { horizontal, vertical };
```

一旦做出某种选择，它也就确定了在扩展系统时需要做哪些修改。在滚动条的例子中，我们可能想引进第三种滚动条。我们开始时或许认为只可能有两种滚动条（“毕竟窗口只有两个维”）。但是，在当前情况中就像在大部分情况中一样，可能的扩展作为重新设计问题出现了。譬如说，有人可能希望能用一个“浏览按钮”而不是两个滚动条。这种按钮在用户按压其不同位置时导致不同方向的滚动。按上面的中间可能导致“向上滚动”，按中间的左边可能导致“向左滚动”，按左上角可能导致“向左上滚动”。这种按钮并非罕见，它们可以看做是滚动条概念的一种精化，特别适合一些应用，其中被滚动的不是普通文字，而是更一般的不同种类的图形。

将一个浏览按钮加进一个采用三个滚动条类的程序里涉及到加入一个新类，但不需要修改原有的滚动条代码：



这是“层次结构”解决方案的漂亮之处。

将滚动条的方向作为参数传递，就意味着在滚动条对象里有一个类型域，在滚动条的成员函数的代码里使用了switch语句。这就是说，我们将不得不在如何表述系统结构的这个方面上做出一种权衡，是基于声明来表示呢，还是基于代码。前一种方式提高了静态检查的级别，也增加了工具需要处理的信息量；后一种方式将决策推迟到运行中，允许通过修改各个函数面完成改变，不影响由类型检查器或其他工具所看到的系统的整体结构。在大部分情况下，我建议用类层次结构直接模拟概念间的层次结构关系。

单一滚动条类型的解决方案使我们很容易保存和传递有关特定种类的滚动条的信息：

```
void helper(Orientation oo)
{
    // ...
    p = new Scrollbar(oo);
    // ...
}

void me()
{
    helper(horizontal);
    // ...
}
```


这种表示也使得在运行中很容易转变滚动条的方向。在滚动条的例子中，一般说，这种功能并不重要，但对另一些等价的例子就可能很重要了。在这里，要点是总有一些需要权衡的东西，而这些权衡常常是很不简单的。

24.3.4.2 包容/层次结构间的权衡

现在考虑如何将滚动条附着到窗口上。如果我们将`Window_with_scrollbar`看做某种既是`Window`又是`Scrollbar`的东西，我们就有了

```
class Window_with_scrollbar : public Window, public Scrollbar {
    // ...
};
```

这就使任何`Window_with_scrollbar`的行动像`Window`和`Scrollbar`，但它也将我们限制到只能有单个滚动条类型的解决方案中。

在另一方面，如果我们将`Window_with_scrollbar`设想为有`Scrollbar`的`Window`，我们就可以得到

```
class Window_with_scrollbar : public Window {
    // ...
    Scrollbar* sb;
public:
    Window_with_scrollbar(Scrollbar* p, /* ... */): Window(/* ... */), sb(p) { /* ... */ }
    // ...
};
```

这使我们能采用滚动条层次结构解决方案。将滚动条作为参数传递，使窗口的滚动条类型变得很明显，我们甚至可能用类似前面传递`Orientation`的方式（24.3.4.1节）传递`Scrollbar`。如果我们需要`Window_with_scrollbar`能像滚动条一样活动，那么就可以增加一个转换操作：

```
Window_with_scrollbar::operator Scrollbar&()
{
    return *sb;
}
```

我喜欢让窗口包含滚动条。我觉得，与想像一个窗口除了是窗口外还是一个滚动条相比，想像一个窗口有一个滚动条更容易一些。事实上，我最赞赏的涉及滚动条的设计策略是将它作为一类特殊窗口，而后将它们包容在需要滚动条服务的窗口里面。这种策略促使我们偏向于包容式的解决方案。采纳包容方案的另一论据来自“它能有几个？”的经验规则（24.3.4节）。因为不存在任何逻辑原因说窗口不能有两个滚动条（事实上，许多窗口都既有水平滚动条，也有垂直滚动条），所以`Window_with_scrollbar`不应该由`Scrollbar`派生。

注意，从未知的类派生是不可能的，在编译时必须确知基类的类型（12.2节）。在另一方面，如果将类的某个属性作为参数传递给它的构造函数，那么在类中某处就必须有一个表示该属性的成员。当然，如果这个成员是指针或者引用，我们就可以将由该成员确定的类的派生类的对象传入。例如，在前面例子里的`Scrollbar*`成员`sb`可以指向另一个类型的`Scrollbar`，例如`Navigation_button`，而`Scrollbar*`的用户根本不知道这个情况。

24.3.5 使用关系

对于一个类使用另外的哪些类，以及如何使用它们的知识，在表述和理解一个设计时都非常重要。C++只是隐式地支持这种依赖性。一个类只能使用（在某个地方）声明过的名字，

但在C++源代码里却没有提供所使用名字的列表。为提取到这些信息，就需要有工具（或者在缺乏适当工具时，通过认真阅读）。类X使用另一个类Y的方式可以按许多方式进行分类，下面是一种分类方式：

- X使用名字Y。
- X使用Y。
 - X调用Y的某个成员函数。
 - X读Y的某个成员。
 - X写Y的某个成员。
- X创建Y。
 - X分配一个类型Y的*auto*或*static*变量。
 - X用*new*创建一个Y。
- X取用Y的大小。

将取对象的大小单独划为一类是因为做这件事时需要有关类声明的知识，但并不依赖于其构造函数。名字Y也划为单独的一种依赖关系，因为只做这件事（例如，声明一个Y* 或在一个外部函数的声明中提到Y）根本不需要访问Y的声明（5.7节）。

```
class Y; // Y是一个类名
Y* p;
extern Y f(const Y&);
```

区分对一个类的界面（类声明）的依赖关系和对一个类实现（成员函数定义）的依赖关系是非常重要的。在一个设计良好的系统里，通常后者都是更强得多的依赖性，对于用户而言，这种情况也远远不如对类声明的依赖性那么有意思（24.4.2节）。通常，一个设计的目标都是使界面的依赖性最小化，因为这种依赖性将变成类用户的依赖性（8.2.4.1节、9.3.2节、12.4.1.1节、24.4节）。

C++并不要求类的实现者去描述它使用了另外的哪些类以及怎样使用的细节。这样做的一个原因是，一些最重要的类依赖于太多其他的类，简化这些类的列表（例如一个*#include*指令）对于可读性是必须的。另一个原因是，对这种依赖性的分类和粒度控制似乎不应该是程序设计语言的问题。这里宁可将使用依赖性的确切方式看成是与设计师、程序员或者工具的目标有关的。最后，需要考虑哪些依赖关系也可能与语言实现的细节有关。

24.3.6 编入程序里的关系

一种程序设计语言不可能也不应该直接支持每种设计方法里的每一个概念。与此类似，一种设计语言也不应该支持每种程序设计语言里的所有特征。一种设计语言应该比较丰富，且较少关心那些用于系统程序设计的语言所必须适应的细节。与此相对应，一种程序设计语言也必须能支持各种设计哲学，否则的话它就将因为缺乏适应性而失败。

当某种程序设计语言没有提供某些功能来直接表示来自设计的某一概念时，就需要采用一种从设计结构到程序设计语言结构的方便映射。举个例子，某设计方法可能有一种委托概念，也就是说，设计中可以这样描述：每个对类A没有定义的操作都应该由指针p所指向的那个类B的对象提供服务。C++无法直接表示这个概念。但是，在C++里这个想法的表述如此规格化，使人很容易设想一个生成代码的程序。考虑

```

class B {
    // ...
    void f();
    void g();
    void h();
};

class A {
    B* p;
    // ...
    void f();
    void ff();
};

```

对一个要求A通过A::p向B委托的规范，其结果代码大致是如下形式：

```

class A {
    B* p;    // 通过p委托
    // ...
    void f();
    void ff();
    void g() { p->g(); }    // 委托g()
    void h() { p->h(); }    // 委托h()
};

```

对程序员而言，这里发生的情况相当清楚，但在代码中模拟设计概念明显不如一对一的对应关系。这种“编入程序的”关系显然不如程序设计语言“理解”的关系，因此也就难于通过工具去操作。举例说，标准工具不可能识别出从A到B通过A::p的“委托”，无法将它与B*的其他使用区分开。

在任何地方，只要可能，都应该去做从设计概念到程序设计语言概念间的一对一映射。一对一映射保证了简单性，保证设计真正能反应在程序里，也就使程序员和工具可以利用它。

转换运算符为表达一类编入程序的关系提供了一种语言机制。也就是说，转换运算符X::operator Y()描述的是在任何可以接受Y的地方都可以使用X（11.4.1节）。构造函数Y::Y(X)表述的是同一种关系。注意，转换运算符（和构造函数）都是生成新对象，而不是改变现有对象的类型。声明到Y的转换函数就是一种要求隐式调用一个返回Y的函数的方式。因为由构造函数和转换运算符所定义的隐式转换有可能失控，在设计时将它们分离出来进行分析将很有价值。

保证在一个程序的转换图里不存在环是很重要的。如果存在环，所引起的歧义性错误将使涉及环路的类型无法组合在一起使用。例如，

```

class Rational;

class Big_int {
public:
    friend Big_int operator+ (Big_int, Big_int);
    operator Rational();
    // ...
};

class Rational {
public:
    friend Rational operator+ (Rational, Rational);
    operator Big_int();
    // ...
};

```

Rational和**Big_int**类型不会像人们所希望的那样平滑交互：

```
void f(Rational r, Big_int i)
{
    g(r+i);           // 错误：歧义：operator+(r, Rational(i)) 或operator+(Big_int(r), i)?
    g(r+Rational(i)); // 一种隐式转换
    g(Big_int(r)+i);   // 另一种隐式转换
}
```

避免这种相互转换的一种方式就是使其中的某个（或者某些）显式化。例如，将从**Big_int**到**Rational**的转换定义为**make_Rational()**，而不是作为转换运算符，那么上面的那个加法就会被解析为**g(Big_int(r), i)**。在无法回避“相互的”转换运算符之处，我们或者需要如上的显式转换，或者需要为+这样的二元运算符定义许多不同版本。

24.3.7 类内的关系

一个类里可能隐藏几乎任何的实现细节和几乎任意数量的烂泥——有时它必须这样。然而，大部分类的对象本身也有某种正规的结构，可以用很容易描述的方式操作。一个类的对象是另一些子对象（通常称为成员）的汇集，其中许多是到其他对象的指针或者引用。这样，一个对象就可以看做是一棵对象树的根，所涉及的对象可以看做是组成了一个“对象层次结构”，如24.3.2.1节所述，这是对类层次结构的补充。举个例子，考虑一个很简单的**String**：

```
class String {
    int sz;
    char* p;
public:
    String(const char* q);
    ~String();
    // ...
};
```

一个**String**对象可以用下图表示：



24.3.7.1 不变式

成员的值和由成员所引用的对象的值汇集在一起就称为这个对象的状态（或简单说是它的值）。有关类设计的一个最主要考虑就是使对象进入一个定义良好的状态（初始化/构造），在每个操作执行中维护定义良好的状态，直至最后得体地销毁这个对象。使一个对象的状态定义良好的性质被称为它的不变式。

这样，初始化的作用就是将对象置入一个满足不变式的状态中，这件事通常由构造函数完成。类中的每个操作都假定在进入它时不变式为真，且在退出时必须保持不变式的真。析构函数最后通过销毁对象来破坏这个不变式。例如，构造函数**String::String(const char*)**保证**p**将指向一个至少有**sz + 1**个元素的数组，这里**sz**具有合理的值，且**p[sz] == 0**。每一个串操作都必须保持这个断言为真。

在类设计中的大量技巧都涉及到如何将类做得足够简单，使得它的实现具有能够简单表

述的有用的不变式。要说明每个类都需要一个不变式非常简单，难的是做出一个有用的不变式，使它很容易理解，又不会给实现者或者操作的效率强加上无法接受的束缚。注意，这里说的“不变式”指的是一段代码，可以运行它来检查对象的状态。很清楚，可以给出更严格更数学化的概念，在某些环境中那样做也更合适。这里所讨论的不变式则是指实际检查对象的状态——因此，通常是经济的、逻辑上不完整的。

不变式的概念源自Floyd、Naur和Hoare有关前条件和后条件的工作，它出现在过去30来年有关抽象数据类型以及程序验证的几乎所有工作中。它也是C排错的主要问题。

典型情况是，成员函数在执行过程中并不维护不变式。在不变式非法时可能被调用的函数不应该是公用界面的一部分。私用和保护函数可以服务于这一用途。

我们怎么能在C++程序里表述不变式的概念呢？一种简单方法就是定义一个检查不变式的函数，并将对它的调用插入公用操作中。例如，

```
class String {
    int sz;
    char* p;
public:
    class Range {};           // 异常类
    class Invariant {};

    enum { TOO_LARGE = 16000 }; // 长度限制
    void check();             // 不变式检查

    String(const char* q);
    String(const String&);
    ~String();

    char& operator[] (int i);
    int size() { return sz; }

    // ...
};

void String::check()
{
    if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
}

char& String::operator[] (int i)
{
    check();                 // 在入口检查
    if (i<0 || sz<=i) throw Range(); // 工作
    check();                 // 在出口检查
    return p[i];
}
```

这将工作得很好，对程序员而言也不费多少事。不过，对于像String这样简单的类，对不变式的检查有可能会主导执行的时间，甚至主导代码的规模。因此，程序员常常是只在调试期间执行不变式检查：

```
inline void String::check()
{
    #ifndef NDEBUG
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz]) throw Invariant();
    #endif
}
```

这里对**NDEBUG**宏的使用方式与标准C的**assert()**中的使用方式类似。很容易设置**NDEBUG**去表明并没有在做排错。

定义不变式以及在排错过程中使用它们的工作虽然很简单，然而却对代码的正确性很有帮助。更重要的是，这样做能使由类所代表的概念的定义良好且规范。这里的要点是，在你设计不变式时，实际上是从另一个观点去看这个类，也使代码中出现了冗余。这两者都增加了辨认出错误、不一致和疏忽的可能性。

24.3.7.2 断言

不变式是断言的特殊形式。一个断言就是一个陈述，它给出了一个必须成立的逻辑准则。问题是，如果它不成立时应该怎么办。

C标准库（言外之意，C++标准库）在 **<cassert>** 和 **<assert.h>** 里提供了**assert()**宏。**assert()**将去求它的参数的值，如果结果是零（**false**）就调用**abort()**。例如，

```
void f(int* p)
{
    assert(p!=0); // 断言p != 0; 如果p是0就abort()
    // ...
}
```

在执行**abort()**之前，**assert()**将输出它自己所在的源文件名和出现的行号，这些都使**assert()**成为一种很有用的排错辅助功能。**NDEBUG**通常通过编译器选项设置，采用基于编译单位的形式。这意味着不应把**assert()**用到那些将被包含到多个编译单位的在线函数和模板函数里，除非极其小心，保证能一致性地设置**NDEBUG**（9.2.3节）。像所有的宏魔术一样，**NDEBUG**的使用过于低级污秽，也很容易出错。还有，即使是在经过良好测试的程序里，留下一些检查也是个很好的主意，而**NDEBUG**就不适合这件事了。进一步说，在产品代码里调用**abort()**也是无法接受的。

另一种方式是用一个**Assert()**模板，让它抛出一个异常而不是中止程序，这样，在需要的时候就可以将断言留在产品代码里。可惜标准库没有提供**Assert()**。但要定义它也很容易：

```
template<class X, class A> inline void Assert(A assertion)
{
    if (!assertion) throw X();
}
```

Assert()在**assertion**为假时抛出异常**X()**。例如，

```
class Bad_arg { };
void f(int* p)
{
    Assert<Bad_arg>(p!=0); // 断言p != 0; 如果p是0抛出Bad_arg
    // ...
}
```

这种风格的断言要求显式的条件，所以，如果我们想只在排错中检查，我们就必须明确说出来。例如，

```
void f2(int* p)
{
    Assert<Bad_arg>(NDEBUG || p!=0); // 或者我未在排错，或者p != 0
    // ...
}
```

在这个断言里用 `||` 而不是 `&&` 可能看起来有些奇怪, 然而 `Assert<E>(a || b)` 检测的是 `!(a || b)`, 这正是 `!a && !b`。

按这种方式使用 `NDEBUG`, 就要求我们无论是否正在排错都必须用某个适当的值去定义好 `NDEBUG`。按照默认方式, C++ 实现不会为我们做这件事, 所以最好是用一个值。例如,

```
#ifdef NDEBUG
const bool ARG_CHECK = false;           // 我们没在排错, 不能检查
#else
const bool ARG_CHECK = true;            // 我们在排错
#endif

void f3(int* p)
{
    Assert<Bad_arg>(!ARG_CHECK || p!=0); // 或者我没在排错, 或者 p != 0
    // ...
}
```

如果与某个断言相关的异常没有被捕获, 失败的 `Assert()` 就会调用 `terminate()` 去结束程序, 很像等价的 `assert()` 将调用 `abort()`。当然, 一个异常处理器就可以采取某种不那么剧烈的行动。

在任何实际规模的程序里, 我发现自己常常会为了测试的需要而成组地打开或者关闭断言。采用 `NDEBUG` 是这种技术中最鲁莽的。在早期开发中, 大部分断言都被启用, 到发布代码时, 就只有那些关键性的稳健检查留在那里, 依然启用。如果实际断言只分为两部分, 这种风格的使用就很容易管理, 只要让第一部分有一个打开条件 (如 `ARG_CHECK`), 而让第二部分始终启用。

如果启用条件是常量表达式, 在不启用时, 相应断言就会被编译丢掉。但是, 在启用条件里也可能需要变量, 以便它能在排错需要时, 在运行中打开或者关闭。例如,

```
bool string_check = true;

inline void String::check()
{
    Assert<Invariant>(!string_check || (p && 0<=sz && sz<TOO_LARGE && p[sz]==0));
}

void f()
{
    String s = "wonder";
    // 在这里串被检查
    string_check = false;
    // 在这里串不检查
}
```

很自然, 这时就会生成代码, 如果广泛使用这种断言, 那就需要保持对代码膨胀的警觉。

比如, 说

```
Assert<E>(a);
```

也就是说

```
if (!a) throw E();
```

的另一种方式。那么为什么还要去用 `Assert()` 而不直接将语句写出来呢? 采用 `Assert()` 使设计师的意图更加明显, 它说这里是断言了某种假定应该总是成立的情况。它并不是程序本身的逻辑的一个组成部分。对于读程序的人而言, 这也是很有价值的信息。另一个实际优点就是,

`Assert()` 或者 `assert()` 很容易查找, 而找到抛出异常的条件语句就不容易了。

可以将 `Assert()` 推广为抛出某些带参数的异常或者变量异常:

```
template<class A, class E> inline void Assert(A assertion, E except)
{
    if (!assertion) throw except;
}

struct Bad_g_arg {
    int* p;
    Bad_g_arg(int* pp) : p(pp) { }
};

bool g_check = true;
int g_max = 100;
void g(int* p, exception e)
{
    Assert(!g_check || p!=0, e); // 指针合法
    Assert(!g_check || (0<*p&&*p<=g_max), Bad_g_arg(p)); // 值可行
    // ...
}
```

在许多程序里, 如果某处的 `Assert()` 可以在编译时求值, 就不应该对它产生代码, 这一点很关键。可惜有些编译器无法对推广的 `Assert()` 做到这一点。因此, 这种两参数的 `Assert()` 只应该用在那种异常的形式不是 `E()`, 而且无论断言的值如何, 所产生出代码都可以接受的地方。

在23.4.3.5节提出了两种常见的类层次结构重组形式, 将一个类分裂为二, 以及从两个类提取出公共部分形成一个基类。在这两种情况下, 良好定义的不变式都能给出重组的可能性的线索。对于那些应该分裂的类, 将不变式与操作代码比较, 将显示出大部分不变式检查是多余的。在这些情况下, 操作的某些子集合都只访问对象状态的子集。相反, 应当归并的类将带有类似的不变式, 即使其详细的实现存在差异。

24.3.7.3 前条件和后条件

断言的一种很流行的应用是用于描述函数的前条件和后条件, 也就是说, 检查对函数输入应该成立的基本假设, 验证在函数离开时留下的是符合预期的状态。可惜的是, 与程序设计语言允许我们方便而有效表达的情况相比我们想做的断言常常位于比更高的层次。例如,

```
template<class Ran> void sort(Ran first, Ran last)
{
    Assert<Bad_sequence>("[first,last) is a valid sequence"); // 伪代码
    // .. 排序算法...
    Assert<Failed_sort>("[first,last) is in increasing order"); // 伪代码
}
```

这一问题具有根本性, 我们关于程序想说的事情最好是用某个基于数学的更高层次的语言表述, 而不是在我们写程序的这种算法式的程序设计语言中表述。

与不变式一样, 要将我们想断言的想法翻译为某种能通过算法去检查的东西, 需要有一点智慧。例如,

```
template<class Ran> void sort(Ran first, Ran last)
{
    // [first,last) 是合法序列: 检查基本要求
    Assert<Bad_sequence>(NDEBUG || first<=last);
```



```
// ...排序算法...
// [first, last) 递增: 检查一个样例
Assert<Failed_sort> (NDEBUG ||
    (last-first<2 || (*first<=last[-1]
        && *first<=first[(last-first)/2] && first[(last-first)/2]<=last[-1])));
}
```

我常常发现, 写出常规的参数检查代码, 得到的结果比组合出的断言还简单。当然, 重要的是试着去表述实际的 (理想的) 前条件和后条件, 至少是将它们写为注释, 在将它们简化到不那么抽象的、能够在程序语言里有效地表达的东西之前这样做。

前条件检查很容易蜕化为对参数值的简单检查。参数常常传过多个函数, 使这一检查重复进行并带来很大的开销。还有, 在每个函数里的每一点都简单陈述参数非零不会有特别的帮助, 而且会给人一种虚假的安全感觉——特别是如果只在排错期间测试, 以避免运行开销。这也是我特别提出应该关注不变式的主要原因。

24.3.7.4 封装

注意, 在C++里, 类是封装的单位, 而单个的对象不是。例如,

```
class List {
    List* next;
public:
    bool on(List*);
    // ...
};

bool List::on(List* p)
{
    if (p == 0) return false;
    for (List* q = this; q; q=q->next) if (p == q) return true;
    return false;
}
```

在私用的`List::next`上寻觅是可以接受的, 因为`List::on()`可以访问它能引用到的类`List`的每个对象。在这样做不方便的地方, 也可以不利用这种从一个成员函数里访问其他对象的内部表示的能力。例如,

```
bool List::on(List* p)
{
    if (p == 0) return false;
    if (p == this) return true;
    if (next==0) return false;
    return next->on(p);
}
```

不过, 这个做法使迭代变成了递归。如果编译器不能将这个递归优化回到迭代的话, 这种做法就会带来显著的性能损害。

24.4 组件

设计的单位是一批类、函数等的集合, 而不是单个孤立的类。这样一个集合常常被称为一个库或者一个框架 (25.8节), 它也是重用 (23.5.1节) 和维护等工作的单位。C++提供了三种主要机制, 用以表述根据逻辑准则形成单位的一组功能的观念:

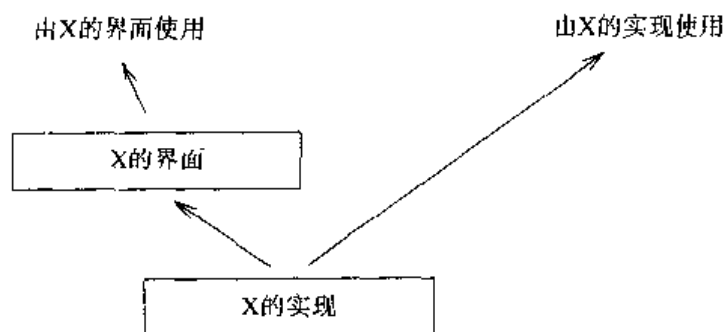
[1] 类——包含一组数据、函数、模板和类型成员。

[2] 类层次结构——包含一组类。

[3] 名字空间——包含一组数据、函数、模板和类型成员。

一个类提供了许多功能，通过它能方便地创建由它所定义的类型对象。但是，也有许多重要成分的表述最好不要采用创建某种单一类的对象的方式。一个类层次结构表述的是一组相互有关的类型。但是，表述一个组件的各个成员的最佳方式未必就是类，也不是所有的类都具有所需要的相似性，能放入某个有意义的类层次结构里（24.2.5节）。因此，名字空间就成为在C++里组件概念的最直接最一般的体现。组件也常常被称做“类范畴（class category）”。当然，并不是说组件里的每个成员都是类。

理想情况下，一个组件的描述包括它所使用的一组界面，再加上由它提供给用户的一组界面。除此之外的所有东西都是“实现细节”，对系统其他部分是隐藏的。这可能就是设计师的界面描述。为使它成为现实，程序员需要将它们映射到一些声明。类和类层次结构提供了各种界面，名字空间使程序员可以将界面组织起来，将所使用的界面与所提供的界面分开。考虑



采用8.2.4.1节的技术，这将变成

```

namespace A { // 由X的界面使用的某些功能
    // ...
}

namespace X { // 提供X的界面

    using namespace A; // 依赖于A里的声明
    // ...
    void f();
}

namespace X_impl { // X的实现所需要的功能
    using namespace X;
    // ...
}

void X::f()
{
    using namespace X_impl; // 依赖于X_impl的声明
    // ...
}
  
```

一般的界面X不应依赖于实现界面X_impl。

一个组件里可能包含了许多其本意并不是为了一般使用的类，这些类应该“隐藏”在实现类或者名字空间之内：

```
namespace X_impl { // 组件X的实现细节
    class Widget {
        // ...
    };
    // ...
}
```

这就保证了 **Widget** 不能被程序的其他部分使用。当然，代表具体概念的那些类通常都是重用的候选者，因此应该考虑包含在组件的界面之中。考虑

```
class Car {
    class Wheel {
        // ...
    };
    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

在大部分环境中，为了保持汽车的抽象性，我们都应该隐藏起实际的车轮（当你使用汽车时不会独立地去操作车轮）。然而，**Wheel** 类本身看起来还是能广泛使用的很好候选者，因此，将它移到 **Car** 类之外可能更好些：

```
class Wheel {
    // ...
};

class Car {
    Wheel flw, frw, rlw, rrw;
    // ...
public:
    // ...
};
```

是否嵌入内部的决策依赖于设计目标以及所涉及概念本身的一般性。嵌入内部和“非嵌入”都是在表达设计时广泛使用的技术。默认选择应该是尽可能使类局部化，直到出现某种需要，说明应该将它做得更为广泛可用。

存在着一种讨厌的倾向，想将“有趣的”函数和数据“浮起”到全局名字空间、广泛使用的名字空间、层次结构中最终的基类里。这样就很容易导致实现细节的并非有意的暴露，并导致与全局数据和全局函数类似的问题。在单根层次结构中，以及只使用了少数几个名字空间的程序里，最容易出现这种情况。在类层次结构中可以用虚基类抵御这种现象。小的“实现性”名字空间是在名字空间的环境里避免这种问题的主要手段。

注意，头文件是一种强有力机制，可以用于针对不同用户提供有关组件的不同观点，也可用于将某些作为细节的类从用户的视线中消除掉。

24.4.1 模板

从设计的角度看问题，模板支持着相互间关系并不密切的两种需要：

— 通用型程序设计。

一 策略的参数化。

在设计的早期，操作仅仅是操作。而到了后来，到了需要描述操作对象的类型时，对于静态类型的程序语言如C++而言，模板就变得非常重要了。没有模板，就需要写出重复的函数定义，或者需要将检查毫无必要地推迟到运行时去做（24.2.3节）。如果一个操作实现的是一个能用于多种类型的算法，它就是实现为模板的候选。如果所有操作对象都能放入一个类层次结构，特别是如果需要在运行时加入新运算对象类型，那么这个运算对象类型最好是表示为一个类——常常是一个虚基类。如果运算对象类型不能放入一个类层次结构里，特别是如果对运行时间有严酷的要求，那么这个操作最好用模板表示。标准容器及其支持算法就是这方面的例子，在这里需要处理各种不相关类型的运算对象，而且有运行性能的需求，所以使用的是模板（16.2节）。

为使在模板/层次结构之间的权衡情况更具体，现在考虑如何推广一个简单的迭代：

```
void print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

这里，假定`Iter_for_T`所提供的操作产生出`T*`。

我们可以将迭代器`Iter_for_T`作为模板的参数：

```
template<class Iter_for_T> void print_all(Iter_for_T x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

这就使我们可以利用各种不相关的迭代器，只要它们提供了具有正确意义的`first()`和`next()`操作，而且在我们编译时对每个`print_all()`调用都能知道迭代器的类型。标准库容器和算法都基于这一思想。

换一种方式，我们也可以采用另一种观点，认为`first()`和`next()`组成了针对迭代器的界面。我们随后就可以定义一个类来表示这个界面：

```
class Iter {
public:
    virtual T* first() const = 0;
    virtual T* next() = 0;
};

void print_all2(Iter& x)
{
    for (T* p = x.first(); p; p = x.next()) cout << *p;
}
```

我们现在就可以使用所有从`Iter`派生的迭代器了。无论我们是用模板还是用类层次结构表示参数化，实际代码都没有什么差别，只有在运行时间、重新编译等的折中方面存在差异。特别是，类`Iter`是作为模板参数的候选：

```
void f(Iter& i)
{
    print_all(i);    // 使用模板
    print_all2(i);
}
```

因此，这两种方式也可以看做是互补的。

模板常常需要用函数和类作为它实现的一部分，要求其中的大多数也必须是模板，以维持通用性和效率。按照这种方式，算法就变成在一个类型范围中的通用型算法了。这一风格的模板应用也被人们称为通用型程序设计（2.7节）。当我们对一个`vector`调用`std::sort()`算法时，该向量的元素就是`sort()`的操作对象，这样，`sort()`对于元素类型就是通用的。此外，标准排序操作对于容器类型也是通用的，因为可以针对任意符合标准的容器的迭代器去调用它（16.3.1节）。

`sort()`算法还对比较准则进行了参数化（18.7.1节）。从设计的角度看，这一点不同于取一个操作并让它的运算对象的类型成为通用的。决定使某个算法针对某个对象（或者函数）参数化，使之能够控制这个算法的行为方式，这是一个更高层次的设计决策。这个决策将使设计师/程序员得以控制某些策略部分，使他们能指挥算法的操作过程。然而，站在程序设计语言的观点上，这里就不存在什么差异。

24.4.2 界面和实现

理想的界面

- 给用户提供了—集完整而和谐的概念。
- 与组件的所有部分保持一致。
- 不将实现细节暴露给用户。
- 能以多种方式实现。
- 可以静态检查。
- 借助于应用层次的类型描述。
- 以有限的且定义良好的方式依赖于其他界面。

在那些将组件的界面提供给外部世界的类之间需要保持一致性，由于注意到这种情况，我们可以简化讨论，只看单个的类就可以了。考虑

```
class Y { /* ... */ };      // X需要
class Z { /* ... */ };      // X需要
class X { // 拙劣界面风格的例子
    Y a;
    Z b;
public:
    void f(const char * ...);
    void g(int[], int);
    void set_a(Y&);
    Y& get_a();
};
```

这个界面存在一些潜在的问题：

- 该界面使用类型`Y`和`Z`的方式使得在编译它时需要知道`Y`和`Z`的声明。
- 函数`X::f()`的参数数目不详、类型未知（有可能是通过由第一个参数提供的“格式串”控制的）。
- 函数`X::g()`有一个`int[]`参数，这或许可以接受，但通常这种情况是抽象层次太低的信号。整数的数组本身没有范围信息，所以它到底有多少元素是不明确的。
- `set_a()`和`get_a()`函数允许直接访问`X::a`，这基本上暴露了类`X`对象的表示。

这些成员函数提供了一个非常低级的抽象。简而言之，具有这种层次的界面的类应属于某个大组件的实现细节——如果它们真的属于某个地方的话。按理想情况，一个界面函数的参数应当带有足够描述它自身的信息。经验规则是，应该能够通过细缆将它们对服务的请求传递给远程服务器。

C++允许程序员将类的表示暴露出来，作为界面的一部分。这种表示也可以被隐藏（使用 *private* 和 *protected*），但编译可以使用它们，以便去分配自动变量，去做函数的在线处理等。这一情况的负面影响是，在类的表示中所使用的类类型可能引起我们所不希望的依赖性。是否采用类型为 *Y* 和 *Z* 的成员要看类型 *Y* 和 *Z* 实际上是什么。如果它们是简单类型，如 *list*、*complex* 或 *string*，对它们的这种使用通常就很合适。可以认为这种类型是稳定的，需要包含它们的类声明是一类可以接受的编译负担。然而，如果 *Y* 和 *Z* 本身又是重要组件的界面类，例如是一个图形系统或者银行账户管理系统的界面类，不直接依赖它们就更明智些。对于这些情况，采用指针或者引用成员就是更好的选择：

```
class Y;
class Z;

class X { // X只通过指针或引用访问Y和Z
    Y* a;
    Z& b;
    // ...
};
```

这样就使 *X* 不再依赖于 *Y* 和 *Z* 的定义了，也就是说，使 *X* 的定义只依赖于 *Y* 和 *Z* 的名字。当然，*X* 的实现将仍然依赖于 *Y* 和 *Z* 的定义，但那将不会对用户产生有害的影响。

这些显示出要点：一个隐藏了大量信息的界面（正如任何有用的界面所应该做的那样）所依赖的东西将远远少于它的实现。举例来说，编译类 *X* 的定义时将不需要访问 *Y* 和 *Z* 的定义。当然，*X* 中那些对 *Y* 和 *Z* 的对象进行操作的成员函数需要访问 *Y* 和 *Z* 的定义。在分析依赖性时，必须分别考虑界面和实现的依赖关系。对于这两种情况，理想都是使系统的依赖图是一个有向无环图，以便对程序理解和测试。当然，与实现相比，这个理想对于界面是更关键的，也是更经常能够达到的。

注意，一个类可能定义了三个界面：

```
class X {
private:
    // 只由成员和友元访问
protected:
    // 只由成员和友元以及
    // 子类的成员和友元访问
public:
    // 一般公开访问
};
```

此外，一个 *friend* 也是公用界面的一部分（11.5节）。

应该把各个成员作为类中尽可能受限的那个界面的一部分。也就是说，成员应该是 *private*，除非有某种原因要求它具有更广泛的可访问性。如果真是这样，那么它就应该是 *protected*，除非有理由要求它是 *public*。将数据成员作为 *public* 或者 *protected* 几乎总是坏主意。组成公用界面的函数和类应该展现出有关一个类的一种观点，适应于它们在表示某个概念中

所扮演的角色。

注意，抽象类可以用于提供更高层次的表示隐藏（12.3节、25.3节）。

24.4.3 肥大的界面

按理想情况，一个界面应该只提供那些有意义的操作，这些操作应该能由实现这个界面的派生类很好地实现。当然，事情并不都那么容易做。考虑表、数组、关联数组、树等等。如第16.2.2节所示，提供这些类型的一个推广（通常称为容器）是很有诱惑力的，有时也是很有用的，因为它可以作为这些类中任何一个的界面。这（明显）能缓解用户必须处理所有这些容器的细节的负担。当然，定义一般容器类的界面很不简单。假定我们要将`Container`定义为一个抽象类型，我们应希望`Container`提供哪些操作呢？我们可以只提供每个容器都能支持的操作（各个操作集合的交集），但这是一个过于狭窄的界面。事实上，对许多有意思的情况，这种交集就是空集。换个方式，我们也可以提供所有操作集合的并集，如果要通过这一界面对某个对象应用一个“不存在的”操作时，就给出一个运行时错误。由这样一组概念的界面并集形成的界面通常被称为肥大的界面。考虑类型T的对象的一种“通用容器”：

```
class Container {
public:
    struct Bad_oper {    // 异常类
        const char* p;
        Bad_oper(const char* pp) : p(pp) { }
    };

    virtual void put(const T*) { throw Bad_oper("Container::put"); }
    virtual T* get() { throw Bad_oper("Container::get"); }

    virtual T*& operator[] (int) { throw Bad_oper("Container::[] (int)"); }
    virtual T*& operator[] (const char*) { throw Bad_oper("Container::[] (char*)"); }
    // ...
};
```

具体的`Container`可以如下声明：

```
class List_container : public Container, private list {
public:
    void put(const T*);
    T* get();
    // ...没有operator[]...
};

class Vector_container : public Container, private vector {
public:
    T*& operator[] (int);
    T*& operator[] (const char*);
    // ...没有put()或get()...
};
```

只要我们当心，一切都会很好：

```
void f()
{
    List_container sc;
    Vector_container vc;
    // ...
    user(sc, vc);
}
```

```

    }
    void user(Container& c1, Container& c2)
    {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // 不要使用c2.get() 或c1[3]
        // ...
    }

```

然而，很少有数据结构能同时很好地支持下标和表风格的操作。因此，描述一个要求这两者的界面可能并不是一个好主意，这样做就会导致采用运行时类型查询（15.4节）或者异常（第14章）去避免运行时的错误。例如，

```

void user2(Container& c1, Container& c2) // 检查很容易，而恢复很困难
{
    try {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    catch(Container::Bad_oper& bad) {
        // 呜呼！
        // 现在怎么办？
    }
}

```

或者

```

void user3(Container& c1, Container& c2) // 早期检查单调乏味，恢复仍然很困难
{
    if (dynamic_cast<List_container*>(&c1) && dynamic_cast<Vector_container*>(&c2)) {
        T* p1 = c1.get();
        T* p2 = c2[3];
        // ...
    }
    else {
        // 呜呼！
        // 现在怎么办？
    }
}

```

在两种情况下，运行时性能都将受损，产生的代码也会很巨大。结果，人们会倾向于忽略潜在错误，盼望着在程序到了用户手中时这些错误不会出现。与此途径有关的问题是穷举测试非常困难且代价高昂。

因此，在运行时性能很重要的地方、要求对代码正确性有较强的保证的地方或者一般说存在替代方式的地方，最好是避免肥大的界面。使用肥大的界面将削弱概念与类之间的对应关系，这样也就打开了将派生仅仅用于实现方便的闸门。

24.5 忠告

- [1] 应该向数据抽象和面向对象程序设计的方向发展；24.2节。
- [2] （仅仅）根据需要使用C++的特征和技术；24.2节
- [3] 设计应与编程风格相互匹配；24.2.1节。

- [4] 将类/概念作为设计中最基本的关注点，而不是功能/处理；24.2.1节。
- [5] 用类表示概念；24.2.1节、24.3节。
- [6] 用继承（仅仅）表示概念间的层次结构关系；24.2.2节、24.5.2节、24.3.2节。
- [7] 利用应用层静态类型的方式给出有关界面的更强的保证；24.2.2节。
- [8] 使用程序生成器和直接界面操作工具去完成定义良好的工作；24.2.3节。
- [9] 不要去使用那些与任何通用程序设计语言之间都没有清晰界面的程序生成器或者直接界面操作工具；24.2.4节。
- [10] 保持不同层次的抽象相互分离；24.3.1节。
- [11] 关注组件设计；24.4节。
- [12] 保证虚函数有定义良好的意义，每个覆盖函数都实现预期行为；24.3.4节、24.3.2.1节。
- [13] 公用界面继承表示的是“是一个”关系；24.3.4节。
- [14] 成员表示的是“有一个”关系；24.3.4节。
- [15] 在表示简单包容时最好用直接成员，不用指向单独分配的对象的指针；24.3.3节、24.3.4节。
- [16] 设法保证使用依赖关系为易理解的，尽可能不出现循环，而且最小；24.3.5节。
- [17] 对于所有的类，定义好不变式；24.3.7.1节。
- [18] 显式地将前条件、后条件和其他断言表述为断言（可能使用`Assert()`）；24.3.5节。
- [19] 定义的界面应该只暴露出尽可能少的信息；24.4节。
- [20] 尽可能减少一个界面对其他界面的依赖性；24.4.2节。
- [21] 保持界面为强类型的；24.4.2节。
- [22] 利用应用层的类型来表述界面；24.4.2节。
- [23] 将界面表述得使请求可以传递给远程的服务器；24.4.2节。
- [24] 避免肥大的界面；24.4.3节。
- [25] 尽可能地使用`private`数据和成员函数；24.4.2节。
- [26] 用`protected/public`区分开派生类的设计者与一般用户间的不同需要；24.4.2节。
- [27] 使用模板去做通用型程序设计；24.4.1节。
- [28] 使用模板去做算法策略的参数化；24.4.1节。
- [29] 如果需要在编译时做类型解析，请使用模板；24.4.1节。
- [30] 如果需要在运行时做类型解析，使用类层次结构；24.4.1节。

第25章 类的作用

有些东西最好是变一变……
而基本主题则应该永远赞颂。
——Stephen J. Gould

类的种类——具体类型——抽象类型——结点——修改界面——对象I/O——动作——
界面类——句柄——使用计数——应用框架——忠告——练习

25.1 类的种类

C++类是一种程序设计语言结构，它可以服务于多种不同的设计需要。事实上，我发现大部分棘手问题的解决方式都涉及到引进一个新类去表述某个概念，而它在先前的草案设计中是隐而不宣的（而且可能需要删除另外一些类）。类所能扮演的大量变化多端的角色也就引出了各种各样的类，它们能很好地服务于各种特殊目的。本章里将描述若干种典型的类，同时讨论它们的威力和弱点：

25.2节 具体类型。

25.3节 抽象类型。

25.4节 结点。

25.5节 动作。

25.6节 界面。

25.7节 句柄。

25.8节 应用框架。

这些“类的种类”是设计概念而不是语言结构。一种不可企及的（或许是不可企及的）理想是有一组最小而相互独立的种类，从它们出发，可以构造出所有行为良好的有用的类。重要的是应该注意到，这里的每一种类在设计中都有其位置，且对于用户而言，没有任何一种能在本质上优于其他种类。关于设计和编程，最令人摸不着头脑的讨论来自某些人企图只排他性地使用某一两种特定的类。这通常是在简单性名义下做的事情，然而结果却是对某些最热衷的类形式的扭曲而不自然的使用。

这里的描述将强调这些种类的类的最纯粹形式。很自然，也可以使用一些混成的形式。不过，一种混成形式应该表现为某个设计决策的结果，基于对各种工程折中形式的评价，而不是由于某些误导的意念，或避免设计决策的产物。“推迟决策”经常不过是“逃避思考”的一种委婉说法。新手设计师通常最好是通过避免使用混成的东西，通过遵循某个现存组件在性质方面的风格，让新组件具有类似的所需要的性质。只有经验丰富的程序员才应该企图去写通用的组件或者库，而且每一个库的设计师都应该“被宣判”在几年里使用他或她的制品，

为其写文档，做维护。另请参见23.5.1节。

25.2 具体类型

如`vector`（16.3节）、`list`（17.2.2节）、`Date`（10.3节）和`complex`（11.3节、22.5节）等的类都是具体的，意思是说它们中的每一个都代表了一个相对简单的概念，带着支持此概念的一组最基本操作。还有，它们的界面与实现都有一对一的对应关系，都不想作为派生的基类。典型情况下，具体类型不适合放进某个相关类的层次结构中，每个具体类型都能够独立地理解，极少需要参考其他的类。如果一个具体类型实现得很好，用它写出的程序在规模和速度上都可以与用户通过手工写出，只实现了有关概念的特殊形式的程序相媲美。类似地，如果其实现有了重大改变，通常就需要修改其界面以反应这种变化。总而言之，一个具体类型就像是一个内部类型。自然，内部类型也都是具体的。用户定义具体类型（如复数、矩阵、错误消息、符号引用等）通常也就是为某些应用领域提供的基本类型。

类界面的确切性质也决定了实现中的哪些改变在这个环境中具有重要意义。更抽象的界面能为实现的修改留下更大的空间，但也可能损失一些运行时的效率。进一步说，一种很好的实现将只依赖于它绝对需要依赖的那些类，不会因为需要适应程序里其他“类似的”类而给这个类的使用带来任何编译时或者运行时的额外开销。

总结起来，一个提供了具体类型的类的目标是：

- [1] 尽可能紧密地与某个特定概念或者实现策略相匹配。
- [2] 通过在线处理、让操作能够尽可能地利用概念及其实现的性质，提供相当于“手工打造”代码的运行时间与空间效率。
- [3] 对其他类只有最小的依赖性。
- [4] 能够独立地理解和使用。

这样做的结果是用户代码与类实现代码的一种紧密的约束。如果类的实现以任何方式改变，用户代码也必须重新编译，因为用户代码里几乎总包含着对在线函数的调用或具体类型的局部变量。

“具体类型”这个名字的选择也是为了与常用术语“抽象类型”对应。具体类型与抽象类型之间的关系将在25.3节讨论。

具体类型不能直接表述共性。例如，`list`和`vector`提供了类似的操作集合，能够被某些模板函数互换式地使用；然而，在类型`list<int>`与`vector<int>`之间，或者`list<Shape*>`和`vecotr<Shape*>`之间没有任何关系（13.6.3节），虽然我们能辨认出它们的相似性。

对于朴素设计的具体类型，这些常常意味着以类似方式去使用它们的代码看起来却很不一样。例如，用`next()`操作迭代通过`List`与用下标迭代通过`Vector`的形式截然不同：

```
void my(List& sl)
{
    for (T* p = sl.first(); p; p = sl.next()) { // “自然的”表迭代
        // 我的代码
    }
    // ...
}

void your(Vector& v)
{
```

```

    for (int i = 0; i < v.size(); i++) {    // “自然的” 向量迭代
        // 你的代码
    }
    // ...
}

```

由于取下一元素操作是表概念的基本操作（但对向量就不那么常见），而下标操作是向量概念的基本操作（但却不是表的基本操作），出现这种迭代风格的差异也就很自然了。与所选用的实现策略“自然”相关的可用操作对于效率而言常常是至关重要的，也与写代码的容易程度密切相关。

在这里明显的暗礁是，本质上类似的操作的代码（例如上面的两个例子）看起来可能并不类似，而且，使用不同类似操作的具体类型就无法互换性地使用了。在实际例子中，人们需要通过许多思考去发现其中的相似性，一旦发现了，还需要通过大量的重新设计，以便提供某些方式来利用这些相似性。标准容器和算法就是一个例子，这里通过透彻的反复思考，才使得在利用具体类型间相似性的同时又不丧失效率和优雅形式成为可能（16.2节）。

为了以某个具体类型作为参数，函数就必须把确切的具体类型作为参数类型。这里并不存在继承关系，无法利用继承关系去减少参数类型的特殊性。正因为这些，如果企图利用具体类型间的相似性，就会涉及到模板和通用型程序设计，如3.8节所述。在使用标准库时，迭代将变成

```

template<class C> void ours(const C& c)
{
    for (C::const_iterator p = c.begin(); p != c.end(); ++p) { // 标准库迭代
        // ...
    }
}

```

这利用了容器之间最基本的相似性，从而也打开了进一步利用它们的可能性，正如标准算法所做的那样（第18章）。

要想用好某个具体类型，用户必须理解它的各种特殊细节。在一个库里（通常）不存在对所有具体类型都成立的最一般性质。而没有这一类可以依赖的性质，用户就必须去了解各个类的情况。这就是为运行时的紧凑性和效率付出的代价。有时付出这种代价很划算，有时就不是这样。也可能有某种情况，其中个别的具体类比更一般的（抽象）类更容易理解和使用。那些表示众所周知的数据类型（例如数组和表）的类尤其是这样。

然而，也应注意，这里的理想仍是隐藏起尽可能多的细节，而又不严重地损害效率。在线函数在这方面极有价值。通过公用的方式将变量暴露出来，或者提供存取函数使用户可以直接操作它们都不是好主意（24.4.2节）。具体类型也应该是类型，而不是一口袋二进制位再加上几个为方便使用而提供的函数。

25.2.1 具体类型的重用

具体类型很少被用于作为派生的基类。每个具体类型的目标就是给出对某一独立概念的清晰而有效的表述。将这件事情做得很好的类极少能成为基类的候选，不大会用于通过公用派生去建立不同的而又与之相关的类。这种类更经常被作为成员或者私用基类，在那些情况下可以有效地利用它们，又不会使其界面和实现与新类的界面和实现混和在一起，相互连累。

考虑由**Date**派生出一个新类：

```
class My_date : public Date {
    // ...
};
```

将**My_date**当做普通**Date**使用合法吗？当然，这要依赖于**My_date**到底是什么。但按照我的经验，极少能看到一个具体类型不经修改就能作为很好的基类的情况。

具体类型是以不修改的方式“重用的”，就像**int**一类内部类型（10.3.4节）。例如，

```
class Date_and_time {
private:
    Date d;
    Time t;
public:
    // ...
};
```

这种形式的使用（重用？）通常很简单、有效，效率也很高。

或许，原来没有把**Date**设计为易于通过派生修改的形式是个错误？有时人们说，每个类都应该是开放的，允许通过覆盖和通过派生类的成员函数访问来修改。按此观点可以引导出下面这样的**Date**变形：

```
class Date2 {
public:
    // 公用界面，基本上由虚函数组成
protected:
    // 其他实现细节（可能包含某些表示）
private:
    // 表示和其他实现细节
};
```

为了能比较容易而有效地写出覆盖函数，可以将表示部分声明为**protected**。这样就达到了目标，使**Date2**能通过派生随意塑造而又保持了它的用户界面不变。然而这也有代价：

- [1] 效率较低的基本操作。一次C++虚函数调用比一次常规函数调用慢一个百分比，虚函数不能像非虚函数那样做成在线的，有虚函数的类通常引起每个对象一个机器字的额外空间开销。
- [2] 需要使用自由存储。**Date2**的目标是使由它派生出的不同类的对象得以互换地使用。因为这些派生类的大小可能不同，一件很明显的事情就是需要在自由存储中分配它们，并通过指针或引用去访问它们。这样，真正局部变量的使用就会急剧减少。
- [3] 对用户的不方便性。为利用虚函数所提供的多态性，对**Date2**的访问必须通过指针或者引用进行。
- [4] 较弱的封装。虚函数可以覆盖，保护数据可以从派生类中操作（12.4.1.1节）。

当然，这些代价并不总是影响重大的，按照这种方式定义的类的行为常常就是我们所希望的（25.3节、25.4节）。但是，对于像**Date2**这样简单的具体类型，这些代价根本没有必要，也可能产生重大影响。

最后，一个设计良好的具体类型常常可以用做另一个易塑造的类型的表示。例如，

```
class Date3 {
public:
```

```

    // 公用界面，基本上由虚函数组成
private:
    Date d;
};

```

这就是在必要时将具体类型（包括内部类型）纳入某个类层次结构的方式，另见25.10[1]。

25.3 抽象类型

要松弛一个类的用户与实现者之间的联系，以及创建对象的代码和使用这种对象的代码之间的联系，最简单的方法就是引入一个抽象类，用它表示某一公共概念的一组实现的界面。考虑一个简单的Set：

```

template<class T> class Set {
public:
    virtual void insert(T*) = 0;
    virtual void remove(T*) = 0;

    virtual int is_member(T*) = 0;

    virtual T* first() = 0;
    virtual T* next() = 0;

    virtual ~Set() {}
};

```

这就定义了一个集合的界面，带有一个内部的对元素迭代的概念。缺少构造函数但却有析构函数的情况也很典型（12.4.2节）。可能做出几个不同实现（16.2.1节），例如，

```

template<class T> class List_set : public Set<T>, private list<T> {
    // ...
};

template<class T> class Vector_set : public Set<T>, private vector<T> {
    // ...
};

```

抽象类提供的是这些实现的公共界面。这就意味着我们可以使用某个Set，而不必知道它到底用的是哪种实现。例如，

```

void f(Set<Plane*>& s)
{
    for (Plane** p = s.first(); p; p = s.next()) {
        // 我的某些东西
    }
    // ...
}

List_set<Plane*> sl;
Vector_set<Plane*> v(100);

void g()
{
    f(sl);
    f(v);
}

```

对于那些具体类型，我们需要重新设计实现类去表达某些共性，使用模板去利用其中的共性。而在这里，我们就必须设计一个公共界面（这里是Set），但对于实现的那些类而言，除了

需要实现界面的功能之外，再没有别的共性要求了。

进一步说，*Set*的用户也不必知道*List_set*和*Vector_set*的声明，所以用户也就不依赖于这些声明，当*List_set*和*Vector_set*改变时，甚至是引进了*Set*的某个新实现——譬如说*Tree_set*，用户代码都不必重新编译或者做任何修改。所有的依赖性都包含在那些显式使用了由*Set*派生的类的函数里。特别是，假定按常规方式使用头文件，程序员写*f(Set&)*时只需要包含*Set.h*，并不需要*List_set.h*或者*Vector_set.h*。只有在那些需要建立*List_set*或者*Vector_set*的地方，才需要分别包含相应的“实现头文件”。如果引进另一个抽象类来处理创建对象的请求（“一个工厂”，12.4.4节），还可以进一步将实现与实际的类隔离。

界面和实现的分离也意味着另一种情况：若某个操作在一个特定实现上很自然，但却由于不够一般而无法作为界面的组成部分时，就会缺乏对它的访问手段。举例来说，*Set*没有顺序的概念，我们无法在*Set*的界面上支持下标操作，即使正好是用数组实现了一种特殊的*Set*。这也隐含着由于缺乏手工优化而产生的运行时代价。还有，在线化通常都变得不可行了（除了在某个局部环境里，编译器知道实际类型时），而且所有有意思的界面操作都变成了虚函数调用。与具体类型一样，有时值得为抽象类型而付出代价，有时不值得。总而言之，抽象类型的目标是：

- [1] 定义一个概念，其定义方式允许它的多个实现在一个程序中共存。
- [2] 通过虚函数的使用提供合理的运行时间和空间效率。
- [3] 使每个实现对其他类的依赖性最小。
- [4] 能够独立地理解。

并不是说抽象类型就比具体类型更好，它们只是不同而已。用户需要做出困难而又极其重要的权衡和选择。库的提供者可以通过提供两者来避开这个问题，把选择权留给用户。重要的是应该弄清楚一个类属于哪个世界。企图去限制抽象类型的通用性，以便在速度与具体类型媲美的做法通常是要失败的。抽象类型所允诺的就是具有互换性的实现，以及在修改之后不需要大量的重新编译。与此类似，企图让具体类型提供“通用性”以便与抽象类型概念媲美，一般说也会失败。这两个概念是共存的，它们确实必须共存，因为具体类提供了抽象类型的实现，但是绝不要将它们搅和到一起。

除了其直接实现之外，抽象类型通常不想作为别的进一步派生的基类型。派生经常就是提供一个实现。当然，也可以由一个抽象类派生出另一个扩展的抽象类，以这种方式构造出另一个新界面。这个新抽象类又必须通过进一步派生经由某个非抽象类而得以实现（15.2.5节）。

为什么我们不一开始就由*Set*直接派生出*List*和*Vector*，省去引进*List_set*和*Vector_set*类的麻烦呢？换句话说，为什么在我们在可以有抽象类型的时候还要有具体类型呢？

- [1] 高效率。我们希望有*vector*和*list*这样的具体类型，避免由于将实现与界面联系在一起而导致的额外开销（如抽象类带来的那一类额外开销）。
- [2] 重用。我们需要一种“机制”能够拿来“其他地方”设计的类型（如*vector*和*list*），给它们一个新界面，从而使之与新的库或者应用相配合（而不是重写它们）。
- [3] 多重界面。为许多不同的类使用一个单一公共基类将导致肥大的界面（24.4.3节）。通常更好的方式是服务于新用途的类提供新的界面（例如一个对*vector*的*Set*界面），而不是修改其界面以服务于多种用途。

很自然，这些要点是相互联系的。在*Ival_box*的例子中（12.4.2节、15.2.5节），以及有关容器

设计的环境中(16.2节)都讨论了这些问题的一些细节。采用**Set**基类,结果将得到一个依赖于结点类(25.4节)的基于容器的解决方案。

25.4 结点

带着派生观点去构造类层次结构与用于抽象类型的界面/实现观点很不一样。在这里是把一个类看做整个构造的基础。即使它是一个抽象类,通常其中也提供了一些表示,并为其派生类提供某些服务。结点类的实例如**Polygon**(12.3节),初始时的**Ival_slider**(12.4.1节)和**Satellite**(15.2节)。

典型情况中,在层次结构里的一个类表示了一个具有普遍性的概念,其派生类可以看成是它的专门化。被设计作为某个层次结构中一个不可缺少的部分的典型类被称为结点类,它需要依赖自己基类的服务去提供一些服务,这就是说,它需要调用基类的成员函数。一个典型的结点类不仅提供了由其基类刻画的界面的一个实现(就像实现类为抽象类型所做的那样),还常常加入自己的新函数,这样就将给出一个更宽的界面。考虑24.3.2节有关交通模拟实例中**Car**的情况:

```
class Car : public Vehicle {
public:
    Car(int passengers, Size_category size, int weight, int fc)
        : Vehicle(passengers, size, weight), fuel_capacity(fc) { /* ... */ }

    // 覆盖来自Vehicle的相关虚函数:

    void turn(Direction);
    // ...

    // 增加Car的特殊函数:

    virtual void add_fuel(int amount); // 一辆汽车的运行需要燃料
    // ...
};
```

最主要的函数包括构造函数,程序员通过它们描述那些与模拟有关的基本性质。还有就是那些使模拟过程可以操作**Car**而不必知道其确切类型的虚函数。可以按照以下方式创建和使用一个**Car**:

```
void user()
{
    // ...
    Car* p = new Car(3, economy, 1500, 60);
    drive(p, bs_home, MH); // 进入所模拟的交通模式
    // ...
}
```

结点类通常都需要一些构造函数,而且常常是比较复杂的构造函数。正是在这里,结点类与抽象类型分道扬镳,因为抽象类型通常极少有构造函数。

对于**Car**的操作通常需要在其实现中使用基类**Vehicle**提供的操作。此外,**Car**的用户也依赖于来自其基类的服务,例如**Vehicle**会提供一些基本函数来处理重量和大小等,所以**Car**就不必自己做了:

```
bool Bridge::can_cross(const Vehicle& r)
{
```



```
if (max_weight < r.weight()) return false;
// ...
```

这就使程序员可以在结点类 *Vehicle* 的基础上创建像 *Car* 和 *Truck* 这样的新类，他们只需要描述并实现与 *Vehicle* 不同的东西。这通常被称为“通过差异做程序设计”或者“通过扩展做程序设计”。

与许多结点类一样，*Car* 本身也是进一步派生的很好候选。举例来说，*Ambulance* 需要一些额外的数据和操作以处理急务：

```
class Ambulance : public Car, public Emergency {
public:
    Ambulance();
    // 覆盖相关的Car虚函数:

    void turn(Direction);
    // ...

    // 覆盖相关的Emergency虚函数:

    virtual void dispatch_to(const Location&);
    // ...

    // 添加Ambulance 的特殊函数:

    virtual int patient_capacity(); // 担架数
    // ...
};
```

对结点类总结如下：

- [1] 在为它的实现以及为它的用户提供服务两方面都需要依靠它的基类。
- [2] 为它的用户提供了一个比它的基类更宽的界面(即，一个包含了更多公用成员函数的界面)。
- [3] 基本上是靠其公用界面中的虚函数(但也不必拘泥于此)。
- [4] 依赖于它的所有(直接与间接)基类。
- [5] 只有在它的基类的环境中才能理解。
- [6] 可以被用做进一步派生的基类。
- [7] 可以用于创建对象。

并不是每个结点类都符合第1、2、6和7点，但大部分都符合。一个不符合第6点的类就比较接近一个具体类型，可以称做具体结点类。例如，可以用一个具体结点类去实现一个抽象类(12.4.2节)，这种类的变量可以静态分配或者在堆栈上分配。这样的类有时也被称为叶类。当然，任何代码只要是通过指针或者引用在有虚函数的类上操作，它就必须考虑到进一步派生类的可能性(或者假定，在没有语言支持的情况下，进一步派生就不会出现)。一个不符合第7点的类比较接近抽象类型，也可以被称为抽象结点类。由于某种不幸的传统，许多结点类都有一些 *protected* 成员，为派生类提供了一个较少受限的界面(12.4.1.1节)。

第4点意味着，要编译一个结点类，程序员就必须包含其所有直接和间接基类的声明以及它们转而依赖的所有声明。这也是结点类与抽象类型截然不同之处。抽象类型的用户并不依赖于实现抽象类型的那些类，在编译时也无需去包含它们。

25.4.1 修改界面

按照定义，结点类是某个类层次结构中的一个部分。并不是一个类层次结构中的每个类

都需要提供同样的界面。特别是一个派生类可能提供更多的成员函数，它的兄弟类提供得可能是完全不同的一组函数。从设计的角度，`dynamic_cast`（15.4节）可以看做是一种询问对象是否提供了所需界面的机制。

作为例子，考虑一个简单的对象I/O系统。用户希望从一个流读入对象，确定它们是否具有所期望的类型，而后使用它们。例如，

```
void user()
{
    // ... 假定打开的文件里保存着一些Shape，ss是附在该文件上的输入流 ...
    Io_obj* p = get_obj(ss); // 从流读入对象
    if (Shape* sp = dynamic_cast<Shape*>(p)) {
        sp->draw(); // 使用Shape
        // ...
    }
    else {
        // 呜呼：在Shape文件里的非Shape
    }
}
```

函数`user()`完全通过抽象类`Shape`处理各种形状，因此就能使用各种形状。`dynamic_cast`的使用是必不可少的，因为对象I/O系统能处理许多类型的对象，而且用户可能无意间打开了某种文件，其中包含的确实都是完好的对象，但用户从来也没听说过。

这个对象I/O系统假定了所读写的对象都是由`Io_obj`派生的。类`Io_obj`必须是多态类型，以使我们能使用`dynamic_cast`。例如，

```
class Io_obj {
public:
    virtual Io_obj* clone() const = 0; // 多态的
    virtual ~Io_obj() {}
};
```

在这个对象I/O系统里，最关键的函数就是`get_obj()`，它由一个`istream`中读入数据，并基于这些数据创建起类的对象。假定在输入流里表示对象的数据以一个标识对象类的字符串作为前缀，`get_obj()`的工作就是读入这个字符串，而后去调用某个能创建和初始化正确的类对象的函数。例如，

```
typedef Io_obj* (*PF)(istream&); // 指向返回Io_obj*的函数的指针
map<string, PF> io_map; // 将字符串映射到创建对象的函数
bool get_word(istream& is, string& s); // 读入一个词到s
Io_obj* get_obj(istream& s)
{
    string str;
    bool b = get_word(s, str); // 将开头的词读入str
    if (b == false) throw No_class(); // io格式问题
    PF f = io_map[str]; // 用str查找函数
    if (f == 0) throw Unknown_class(); // 对str没有匹配
    return f(s); // 从流中构造对象
}
```

在名为`io_map`的`map`里保存着一些对偶，每个对偶是一个名字串和一个函数，这些函数能创

建出具有对应名字的类的对象。

我们可以按普通方式定义类`Shape`，但还要让它由`Io_obj`派生，这是`user()`所需要的：

```
class Shape : public Io_obj {
    // ...
};
```

当然，更有意思的（在许多情况下也更实际）是采用不加改变的`Shape`定义（2.6.2节）：

```
class Io_circle : public Circle, public Io_obj {
public:
    Io_circle* clone() const { return new Io_circle(*this); } // 使用复制构造函数
    Io_circle(istream&); // 从输入流初始化
    static Io_obj* new_circle(istream& s) { return new Io_circle(s); }
    // ...
};
```

这是一个例子，说明怎样能将一个类与某个类层次结构相配合，该结构中有一个抽象类，这个类的构造由于前瞻性不足，没有一开始就做成一个结点类，因此不能满足实际需要（12.4.2节、25.3节）。

`Io_circle(istream&)` 构造函数用来自其`istream`参数的数据去初始化对象。`new_circle()` 函数将被放进`io_map`里，使该对象I/O系统能知道这个类。例如，

```
io_map["Io_circle"] = &Io_circle::new_circle;
```

其他形状也可以类似的方式构造出来：

```
class Io_triangle : public Triangle, public Io_obj {
    // ...
};
```

如果提供这些对象I/O工作台变得单调乏味，写一个模板将可能有所帮助：

```
template<class T> class Io : public T, public Io_obj {
public:
    Io* clone() const { return new Io(*this); } // 覆盖Io_obj:clone()
    Io(istream&); // 从输入流初始化
    static Io* new_io(istream& s) { return new Io(s); }
    // ...
};
```

有了这些，我们就可以定义出`Io_circle`如下：

```
typedef Io<Circle> Io_circle;
```

当然，我们还是需要明确定义`Io<Circle>::Io(istream&)`，因为它必须知道`Circle`的细节。

`Io`模板是一个例子，说明了一种将具体类型放进一个类层次结构的方式。在这里提供了一个句柄，让它作为该层次结构里的一个结点。由其模板参数派生就允许从`Io_obj`出发做强制。不幸的是，这也排除了对内部类型使用`Io`的可能性：

```
typedef Io<Date> Io_date; // 包装起具体类型
typedef Io<int> Io_int; // 错误：不能从内部类型派生
```

这个问题还是可以处理的，可以为内部类型提供另一个单独的模板，或者用一个类去表示一个内部类型（25.10[1]）。

这一简单的对象I/O系统可能无法完成我们所希望的所有工作，但它几乎可以放进一页里，

其中的关键概念已经有许多应用。一般说, 这些技术可以用于完成基于用户所提供的字符串的函数调用, 并且可用于通过运行时类型识别发现的界面去操作类型未知的对象。

25.5 动作

在C++里描述动作的最简单最明显的方式就是写一个函数。然而, 如果一个动作需要延迟, 在执行之前必须传递到“某个地方”, 要求有它自己的数据, 必须能与其他动作组合 (25.10[18, 19]) 等, 那么, 通过一个能执行所需的动作并提供其他服务的类来提供这个动作就很有吸引力。这方面的明显例子包括标准算法中所用的函数对象 (18.4节), 以及*iostream*里所用的操控符 (21.3.4节)。对于前一情况, 实际动作由应用运算符执行; 后一种情况中的动作由 << 和 >> 运算符执行。在*Form* (21.4.6.3节) 和*Matrix* (22.4.7节) 的情况中, 采用组合符类去推迟执行, 直到汇集了能有效执行的足够多的信息为止。

动作类的最常见形式是一种很简单的类, 其中只包含一个虚函数 (名字常常是*do_it*之类):

```
struct Action {
    virtual int do_it(int) = 0;
    virtual ~Action() {}
};
```

有了这个类, 我们就可以写出代码 (例如一个菜单), 使之能够存储起需要在后面再执行的动作, 又不必使用函数指针, 不必了解被调用对象的任何信息, 甚至也不必知道被调用操作的名字。看下面例子:

```
class Write_file : public Action {
    File& f;
public:
    int do_it(int) { return f.write().succeed(); }
};

class Error_response : public Action {
    string message;
public:
    Error_response(const string& s) : message(s) {}
    int do_it(int);
};

int Error_response::do_it(int)
{
    Response_box db(message.c_str(), "continue", "cancel", "retry");

    switch (db.get_response()) {
    case 0:
        return 0;
    case 1:
        abort();
    case 2:
        current_operation.redo();
        return 1;
    }
}

Action* actions[] = {
    new Write_file(f),
```

```

    new Error_response("you blew it again"),
    // ...
};

```

*Action*的用户被完全隔离在有关派生类（如*Write_file*和*Error_response*）的任何信息之外。

这是一种威力强大的技术，具有功能分解背景的人员在使用时需要特别小心。如果让过多的类看起来都像*Action*，那么系统的整体设计可能就已经堕落成某种过度功能性的东西了。

最后，也可以用类对操作进行编码，而后传递到远程的机器中执行，或者存储起来留待将来使用（25.10[18]）。

25.6 界面类

类中最重要的一种就是那些最卑微、最不受重视的界面类。界面类不做多少事情——如果那样做就不是一个界面类了。界面类只是为了某些局部需要而简单地调整某些服务的表现形式。因为从原则上说，不可能在任何时候都能为所有需要提供同样好的服务，为了允许共享而又不迫使所有用户都去穿同样一件枷衣，界面类就不可缺少了。

具有最纯粹形式的界面甚至不产生任何代码。考虑13.5节*Vector*的专门化形式：

```

template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    Vector(int i) : Base(i) {}

    T*& operator[] (int i) { return static_cast<T*&>(Base::operator[] (i)); }

    // ...
};

```

这个（部分的）专门化将不安全的*Vector<void*>* 转换为一族更有用的类型安全的向量类。为了使界面类能够负担得起，在线函数是不可或缺的。像上面的情况里，在线前推函数所做的只是类型调整，不存在任何时间或者空间开销。

很自然，一个代表着由具体类型实现的抽象类型的抽象基类（25.2节）也是界面类的一种形式，25.7节的句柄类也是。但无论如何，在这里我们更关注的是那些没有更特殊的功能，仅仅完成界面调整的类。

现在考虑使用多重继承合并两个界面的问题。如果存在某个名字的冲突，也就是说，两个类对执行完全不同操作的虚函数使用了同样的名字，这时我们该怎么办？举个例子，考虑一个Wild-West电子游戏，其中用户交互通过一个通用窗口类处理：

```

class Window {
    // ...
    virtual void draw(); // 显示图像
};

class Cowboy {
    // ...
    virtual void draw(); // 从枪套中拔枪
};

class Cowboy_window : public Cowboy, public Window {
    // ...
};

```

Cowboy_window 表示游戏中一个牛仔的动画，并处理用户/玩游戏者与这个牛仔角色的交互。我们当然希望用多重继承，而不是将 *Window* 或者 *Cowboy* 声明为成员，因为那样就需要定义许多针对 *Window* 和 *Cowboy* 的服务函数。我们希望能将 *Cowboy_window* 传递给这种函数，而不要程序员做任何特殊的事情。然而，这样做就会引起定义 *Window::draw()* 和 *Cowboy::draw()* 的 *Cowboy_window* 版本的问题。

只能有一个定义在 *Cowboy_window* 里的被称为 *draw()* 的函数。然而，由于那些操作 *Window* 或者 *Cowboy* 的服务性函数根本就没有关于 *Cowboy_window* 的任何知识，*Cowboy_window* 必须同时覆盖 *Cowboy* 的 *draw()* 和 *Window* 的 *draw()*。用一个 *draw()* 去覆盖这两个函数是错误的，因为，除了名字相同外，这两个 *draw()* 函数相互无关，不可能通过一个函数重新定义。

最后，我们还希望 *Cowboy_window* 对于继承来的 *Window::draw()* 和 *Cowboy::draw()* 也有不同的无歧义的名字。

为解决这个问题，我们需要为 *Cowboy* 引进一个额外的类，为 *Window* 引进一个额外的类。这两个类为 *draw()* 函数引进两个新名字，并保证在需要调用 *Cowboy* 或 *Window* 中的函数 *draw()* 时，将去调用具有新名字的函数：

```
class CCowboy : public Cowboy {           // 到Cowboy的界面，重命名draw()
public:
    virtual int cow_draw() = 0;
    void draw() { cow_draw(); }           // 覆盖Cowboy::draw()
};

class WWindow : public Window {           // 到Window的界面，重命名draw()
public:
    virtual int win_draw() = 0;
    void draw() { win_draw(); }           // 覆盖Window::draw()
};
```

现在我们可以从界面类 *CCowboy* 或 *WWindow* 组合起 *Cowboy_window*，并用所需要的效果去覆盖 *cow_draw()* 和 *win_draw()*：

```
class Cowboy_window : public CCowboy, public WWindow {
    // ...
    void cow_draw();
    void win_draw();
};
```

注意，只有两个 *draw()* 函数具有完全一样的参数类型时，这才成为一个严重问题。如果它们的参数类型不同，普通的重载解析规则就能保证不会出问题，即使是相互无关的函数具有同样的名字。

对于界面类的每个使用，人们都可以设想一种特殊的语言扩展，它执行所需要的调整，但效率稍微高一点，形式也更优美一点。然而，界面类的每种使用都不会经常遇到，用特殊语言结构去支持它们将带来无法接受的复杂性负担。特别是，由于合并类层次结构而产生的名字冲突并不常见（与程序员需要写类的情况相比），这种情况倾向于出现在合并由互不相干的文化产生出的层次结构时，例如游戏和窗口系统。合并这种互不相干的类层次结构不是一件容易的事，消解名字冲突将更多的是程序员的问题。其他问题包括互不相干的错误处理、互不相干的初始化、互不相干的存储管理策略等。这里要讨论名字冲突的消解问题，是因为

这种引进带有前推函数的界面类的技术还有许多其他应用，不仅可以用于改变名字，还可以改变参数和返回值类型，引进运行时检查，等等。

因为`CCowboy::draw()`和`WWindow::draw()`都是虚函数，它们就无法通过简单的在线方式优化而去掉。当然，编译器也有可能识别出它们不过是一些前推函数，而后去优化经由它们的调用链，将它们去掉。

25.6.1 调整界面

界面函数的主要用途是调整某个界面，使之能更好地与用户的期望匹配，这样也就把原本可能散布在用户代码里各个地方的代码移到了界面里。举个例子，标准`vector`以0作为下标开始，需要用非0到`size - 1`的下标范围的用户就必须调整他们的用法。例如，

```
void f()
{
    vector v<int>(10);           // 下标范围 [0..9]
    // 假装v的范围是 [1..10]
    for (int i = 1; i <= 10; i++) {
        v[i-1] = 7;             // 记住调整下标值
        // ...
    }
}
```

更好的方式是提供一种具有任意边界的`vector`类：

```
class Vector : public vector<int> {
    int lb;
public:
    Vector(int low, int high) : vector<int>(high-low+1) { lb=low; }
    int& operator[] (int i) { return vector<int>::operator[] (i-lb); }
    int low() { return lb; }
    int high() { return lb+size()-1; }
};
```

这个`Vector`类可以像下面这样使用：

```
void g()
{
    Vector v(1, 10);             // 范围是 [1..10]
    for (int i = 1; i <= 10; i++) {
        v[i] = 7;
        // ...
    }
}
```

与前面例子相比，这样做没有增加任何开销。很清楚，这个`Vector`版本更容易读，写起来也更容易出错。

界面类一般都相当小，做的事情也相当少（按定义）。只要依据不同传统写出的软件需要合作，它们就可能冒出来，因为在这里需要弥合不同规定之间的差异。例如，界面类常常被用于为非C++代码提供C++界面，以及用于将应用代码与库的细节隔离（给使用其他库替代这个库留下可能性）。

界面类的另一项重要应用是提供带检查的或者受限的界面。举例来说，希望一些整数变量只取某规定范围内的值也是很常见的情况，这可以通过如下简单模板（在运行时）强制要求：

```
template<int low, int high> class Range {
    int val;
public:
    class Error { }; // 异常类

    Range(int i) { Assert<Error>(low<=i&&i<high); val = i; } // 见24.3.7.2节
    Range operator=(int i) { return *this=Range(i); }

    operator int() { return val; }
    // ...
};

void f(Range<2, 17>);
void g(Range<-10, 10>);

void h(int x)
{
    Range<0, 2001> i = x;    // 可能抛出Range::Error
    int i1 = i;

    f(3);
    f(17);                  // 抛出Range::Error
    g(-7);
    g(100);                 // 抛出Range::Error
}
```

*Range*模板很容易扩展，使之能处理任意标量类型的范围检查问题（25.10[7]）。

控制对其他类的访问或调整其界面的界面类有时也被称为包装器（wrapper）。

25.7 句柄类

抽象类型提供了界面与其实现之间的一种有效隔离。然而，如25.3节所示，在抽象类型提供的界面与具体类型所提供的实现之间建立起的是—种固定联系。举例说，如果某个迭代器原来约束于一种资源（比如说一个集合），在这个资源耗尽时，就不可能将它重新约束于另一资源（比如说一个流）。

进一步说，除非你通过指针或者引用去操作某个抽象类的实现对象，否则就会丢掉虚函数带来的所有利益。用户代码也可能变得依赖于实现类的细节，因为不知道抽象类型的实际大小就不能做静态分配或在堆栈分配（包括无法接受作为值参数）。使用指针或引用，也意味着存储管理的重担落在了用户的身上。

抽象类方法的另一个限制是一个类对象具有固定的大小。然而，我们有时想用一些类去表示某些概念，而实现这些概念所需的存储量可能变化。

处理这类问题的一种流行技术就是将一个对象分成两个部分：一个提供用户界面的句柄部分，另一个是保存着对象状态的几乎所有信息的表示部分。句柄与表示之间的联系通常用句柄中的一个指针表示。句柄常常保存着比单独的表示指针更多一点的信息，但也多得有限。这也就意味着即使表示发生了变化，句柄的布局一般也能保存稳定。还有就是句柄很小，移动它们相对而言没有什么代价，因此用户就不必再用指针或者引用了。



11.12节的***String***是句柄的简单实例，其中的句柄为相应表示提供了界面、访问控制和存储管理。在这个情况下，句柄和表示都是具体类型，但表示类也经常是抽象类型。

考虑取自25.3节的抽象类型***Set***。应该怎样为它提供一个句柄？这一做法将涉及到哪些利益和代价呢？有了这个集合类，我们可以通过重载 \rightarrow 运算符简单地定义一个句柄：

```
template<class T> class Set_handle {
    Set<T>* rep;
public:
    Set<T>* operator->() { return rep; }
    Set_handle(Set<T>* pp) : rep(pp) { }
};
```

这样做不会给***Set***的使用方式带来明显的影响，你只要简单地将***Set_handle***传来传去，而不必再传***Set&*** 或者***Set**** 了。例如，

```
void f(Set_handle<int> s)
{
    for (int* p = s->first(); p; p = s->next())
    {
        // ...
    }
}

void user()
{
    Set_handle<int> sl(new List_set<int>);
    Set_handle<int> v(new Vector_set<int>(100));

    f(sl);
    f(v);
}
```

通常我们希望句柄所做的事情比提供访问更多一点。举例来说，如果***Set***类及其***Set_handle***类是一起设计的，那么就很容易为每个***Set***增加一个计数器，以便做引用的计数。一般来说，我们并不希望与被句柄关联的类一起设计句柄，因为那样我们将不得不把句柄需要存储的某些信息存入另一个对象里。换句话说，我们希望有非侵入式的句柄，而不是侵入式的。例如，下面就是一个句柄，在对象的最后一个句柄离开时，该对象就会被删除：

```
template<class X> class Handle {
    X* rep;
    int* pcount;
public:
    X* operator->() { return rep; }

    Handle(X* pp) : rep(pp), pcount(new int(1)) { }
    Handle(const Handle& r) : rep(r.rep), pcount(r.pcount) { (*pcount)++; }

    Handle& operator=(const Handle& r)
    {
```

```

        if (rep == r.rep) return *this;
        if (--(*pcount) == 0) {
            delete rep;
            delete pcount;
        }
        rep = r.rep;
        pcount = r.pcount;
        (*pcount)++;
        return *this;
    }

    ~Handle() { if (--(*pcount) == 0) { delete rep; delete pcount; } }

    // ...
};

```

这种句柄可以自由地传来传去，例如，

```

void f1(Handle<Set>);

Handle<Set> f2()
{
    Handle<Set> h(new List_set<int>);
    // ...
    return h;
}

void g()
{
    Handle<Set> hh = f2();
    f1(hh);
    // ...
}

```

在这里，由f2()创建的集合将在g()退出时删除——除非f1()保留下另外的副本。程序员根本不必去知道它。

很自然，这种方便也要付出代价，但对许多应用而言，存储和维护引用计数值的代价还是可以接受的。

有时人们需要从句柄里提取出指向表示的指针，并直接去使用它。举例来说，在将对象传递给一个不知道句柄的函数时，就需要采用这种方式。这种做法能够工作得很好，条件是在被调用函数里不销毁传递给它的对象，也不在某个地方存储指向它的指针以便返回到它的调用者之后再用。将句柄另行约束到一个新表示也是一种很有用的操作：

```

template<class X> class Handle {
    // ...

    X* get_rep() { return rep; }

    void bind(X* pp)
    {
        if (pp != rep) {
            if (--*pcount == 0) {
                delete rep;
                *pcount = 1;           // 重复使用pcount
            }
            else
                pcount = new int(1);   // 新的pcount
            rep = pp;
        }
    }
};

```

```

    }
}
};

```

注意,从`Handle`派生出新类不会有什么用处,因为它是个没有虚函数的具体类型。有一个想法是为由一个基类定义出的一族类做一个句柄类。由这样的基类派生可以成为一种非常强有力的技术,这种技术除了能应用于抽象类型之外,也可以应用于结点类。

按照写出的形式,`Handle`并没有处理继承问题。为了得到一个能够像一个真正的引用计数指针那样活动的类,还需要将`Handle`与13.6.3.1节的`Ptr`结合(25.10[2])。

如果一个句柄类所提供的界面接近或者等同于以它作为句柄的那个类,这种句柄通常被称为代理。要引用位于远程计算机上的对象,常常需要使用这种形式的句柄。

25.7.1 句柄上的操作

覆盖运算符 `->` 使句柄可以取得控制,并在每次对象访问时做一些事情。例如,你可以做有关通过一个句柄访问对象的次数统计:

```

template <class T> class Xhandle {
    T* rep;
    int no_of_accesses;
public:
    T* operator->() { no_of_accesses++; return rep; }
    // ...
};

```

如果句柄需要在访问之前和之后都做些事情,那就需要更精细的程序设计。举个例子,假定某人希望在做插入或者删除时设置一个锁。从本质上说,就需要在句柄中重新做表示类的界面:

```

template<class T> class Set_controller {
    Set<T>* rep;
    Lock lock;
    // ...
public:
    void insert(T* p) { Lock_ptr x(lock); rep->insert(p); } // 见14.4.1节
    void remove(T* p) { Lock_ptr x(lock); rep->remove(p); }

    int is_member(T* p) { return rep->is_member(p); }

    T get_first() { T* p = rep->first(); return p ? *p : T(); }
    T get_next() { T* p = rep->next(); return p ? *p : T(); }

    T first() { Lock_ptr x(lock); T tmp = *rep->first(); return tmp; }
    T next() { Lock_ptr x(lock); T tmp = *rep->next(); return tmp; }
    // ...
};

```

提供这些前推函数确实很烦人(而且有时也容易出错),尽管它们都不难做,而且也不会消耗运行时间。

注意,只有某些`Set`函数需要加锁。在我的经验中,在一个类中,只对其中的某些成员函数需要前后动作是很典型的情况。对于加锁的情况,对所有操作都加锁(有些系统的监控程序就是这样做的)将导致过多的锁定,有时会严重影响并发性。

与重载句柄中的 `->` 相比,精细地定义句柄类中所有操作还有另一个优点:有可能从类

*Set_controller*出发进行派生。可惜的是，如果在派生类里增加了数据成员，作为句柄的某些益处就会丧失。特别是与各个句柄中的代码量相比，共享代码的量（在句柄类里）将会减少。

25.8 应用框架

通过25.2节到25.7节所描述的各种类构造出的组件能支持设计和代码的重用，采用的方法是提供构造块和组合它们的方式；应用的构造者们设计出框架，使这些公共的建筑块可以被纳入其中。另一种更加雄心勃勃的支持设计和重用的方法是提供一些能建立起某种公共框架的代码，使应用的构造者们能将有关应用的专用代码作为块装进去。这样的一种途径通常被称为一个应用框架。建立起这种应用框架的类通常都有很肥大的界面，它们很难看做是传统意义下的类型。一个框架很接近一个理想的完整的应用，但是其中什么都不会做。所有的特定操作将由应用程序员提供。

作为例子，下面考虑一个过滤器，也就是说一个程序，它从输入流读，（可能）基于输入去执行某些操作，（可能）产生一个输出流，并（可能）产生一个最后的结果。对于这种程序的一个简单框架应提供一组应用程序员可以提供的操作：

```
class Filter {
public:
    class Retry {
    public:
        virtual const char* message() { return 0; }
    };

    virtual void start() { }
    virtual int read() = 0;
    virtual void write() { }
    virtual void compute() { }
    virtual int result() = 0;

    virtual int retry(Retry& m) { cerr << m.message() << '\n'; return 2; }

    virtual ~Filter() { }
};
```

所有派生类必须提供的函数都被声明为纯虚函数，其他函数都被简单定义成在其中什么也不做的函数。

框架还提供了主循环和一个初步的错误处理机制：

```
int main_loop(Filter* p)
{
    for(;;) {
        try {
            p->start();
            while (p->read()) {
                p->compute();
                p->write();
            }
            return p->result();
        }
        catch (Filter::Retry& m) {
            if (int i = p->retry(m)) return i;
        }
        catch (...) {
            // ...
        }
    }
}
```

```

        cerr << "Fatal filter error\n";
        return 1;
    }
}

```

最后，我将把我的程序写成

```

class My_filter : public Filter {
    istream& is;
    ostream& os;
    int nchar;
public:
    int read() { char c; is.get(c); return is.good(); }
    void compute() { nchar++; }
    int result() { os << nchar << " characters read\n"; return 0; }

    My_filter(istream& ii, ostream& oo) : is(ii), os(oo), nchar(0) { }
};

```

并以

```

int main()
{
    My_filter f(cin, cout);
    return main_loop(&f);
}

```

的方式将它激活。很自然，如果要为某些重要应用做出一个框架，它必须比这个简单实例提供更多的结构和更多的服务。特别是，一个框架通常都是一种结点类的层次结构，让应用程序员为深层嵌套的内部提供叶结点，这样就可以利用这个层次结构来提供应用之间的共性，提供可重用的服务。框架也将由一个库支持，这个库提供一些有用的类，使程序员可以利用它们去描述自己的动作类。

25.9 忠告

- [1] 应该对一个类的使用方式做出有意识的决策（作为设计师或者作为用户）；25.1节。
- [2] 应注意到涉及不同种类的类之间的权衡问题；25.1节。
- [3] 用具体类型去表示简单的独立概念；25.2节。
- [4] 用具体类型去表示那些最佳效率极其关键的概念；25.2节。
- [5] 不要从具体类派生；25.2节。
- [6] 用抽象类去表示那些对象的表示可能变化的界面；25.3节。
- [7] 用抽象类去表示那些可能出现多种对象表示共存情况的界面；25.3节。
- [8] 用抽象类去表示现存类型的新界面；25.3节。
- [9] 当类似概念共享许多实现细节时，应该使用结点类；25.4节。
- [10] 用结点类去逐步扩充一个实现；25.4节。
- [11] 用运行时类型识别从对象获取界面；25.4.1节。
- [12] 用类去表示具有与之关联的状态信息的动作；25.5节。
- [13] 用类去表示需要存储、传递或者延迟执行的动作；25.5节。
- [14] 利用界面类去为某种新的用法而调整一个类（不修改这个类）；25.6节。

- [15] 利用界面类增加检查, 25.6节。
- [16] 利用句柄去避免直接使用指针和引用; 25.7节。
- [17] 利用句柄去管理共享的表示; 25.7节。
- [18] 在那些能预先定义控制结构的应用领域中使用应用框架; 25.8节。

25.10 练习

1. (*1) 25.4.1节的**Io**模版对内部类型不能工作, 修改它, 使它能工作。
2. (*1.5) 25.7节的**Handle**模板不能反映以它作为句柄的类的继承关系。修改这个类, 使它能反映这种关系, 也就是说, 你应该能用**Handle<Circle>** 给**Handle<Shape>** 赋值, 但不能反过来赋值。
3. (*2.5) 假定有了**String**类, 定义另一个串类, 使之用**String**类作为表示, 并将它的操作都作为虚函数。比较这两个类的性能。试着去找到一个有意义的类, 它的最佳实现方式就是从带有虚函数的串类通过公用派生。
4. (*4) 研究两个广泛使用的库, 将库中的类划分为具体类型、抽象类型、结点类、句柄类和界面类。这里使用了抽象结点类和具体结点类吗? 对于这些库里的类还有更合适的分类方式吗? 是否用到了肥大的界面? 为运行时类型信息提供了什么功能(如果有的话)? 其存储管理策略是什么?
5. (*2) 利用**Filter**框架(25.8节)实现一个程序, 该程序从一个输入流中删除相邻的重复单词, 并将其他的值都从输入复制到输出。
6. (*2) 利用**Filter**框架实现一个程序, 该程序统计输入流中单词出现的频率, 产生(单词, 次数)对的列表作为输出。
7. (*1.5) 写一个**Range**模板, 以检查范围和元素类型作为它的模板参数。
8. (*1) 写一个**Range**模板, 以检查范围作为其构造函数的参数。
9. (*2) 写一个不执行错误检查的简单字符串类。写出另一个检查对第一个类的访问的类。讨论将基本函数与错误检查分开的优点和缺点。
10. (*2.5) 为若干其他类型实现25.4.1节的对象I/O系统, 其中至少应包括整数、字符串和你选择的某个类层次结构。
11. (*2.5) 定义类**Storable**为一个带有虚函数**write_out()**和**read_in()**的抽象基类。为简单起见, 假定一个字符串足以表示一个永久存储位置。用类**Storable**提供一种功能, 使我们能将由**Storable**派生的类的对象写进磁盘, 并能从磁盘读入这些对象。用一些你所选择的类去测试它。
12. (*4) 定义一个带有操作**save()**和**no_save()**的基类**Persistent**, 用它可以控制在析构函数里是否将对象写入永久存储器。除了**save()**和**no_save()**之外, 类**Persistent**还应该提供哪些有用的操作? 用一些你选定的类测试**Persistent**类。**Persistent**是一个结点类、具体类型或抽象类型吗? 为什么?
13. (*3) 写一个类**Stack**, 使它可能在运行中改变实现。提示: “任何问题都可以通过另一个间接来解决”。
14. (*3.5) 定义类**Oper**, 它保存着类型为**Id**(可能是**string**或者C风格的字符串)的标识符和一个操作(指向函数或者某种函数对象的指针)。定义类**Cat_object**, 它保存着一个**Oper**的表

和一个`void*`。为`Cat_object`提供函数`add_oper(Oper)`，它能向`Cat_object`的列表中添加一个`Oper`；提供`remove_oper(Id)`，它从`Cat_obj`的列表中删除`Id`所对应的`Oper`；还有`operator()` (`Id, arg`)，它调用`Id`所对应的`Oper`。用`Cat_object`实现一个`Oper`的堆栈。写一个小程序试验这些类。

15. (*3) 基于类`Cat_object`定义一个`Object`模板。用`Object`实现一个`String`的堆栈。写一个小程序试验这个模板。
16. (*2.5) 定义`Object`模板的一个变形，称其为`Class`，它保证带有相同操作的对象共享同一个操作表。写一个小程序试验这个模板。
17. (*2) 定义一个`Stack`模板，使它为通过`Object`模板实现的堆栈提供一种方便的类型安全的界面。用这个堆栈与前面练习中的堆栈做一些比较。写一个小程序试验这个模板。
18. (*3) 写一个类，表示将要发送到另一台计算机上并在那里执行的操作。测试它，通过实际将命令送到另一台计算机，或者将命令写入一个文件而后执行从该文件读入的命令。
19. (*2) 写一个类，它用于组合以函数对象表示的操作。给定了两个函数对象`f`和`g`，`Compose(f, g)`应做出一个对象（该对象可以以适合于`g`的参数`x`调用）并返回`f(g(x))`，只要`g()`返回值的类型可以被接受作为`f()`的参数。

附录和索引

这些附录给出了C++的语法、一个有关C++与C以及标准C++与以前的C++之间兼容性的讨论，还有这个语言的各种技术细节。索引是范围广泛的，也应认为是本书的一个有机组成部分。

章目

- 附录A 语法
- 附录B 兼容性
- 附录C 技术细节
- 附录D 现场
- 附录E 标准库的异常时安全性
- 索引

附录A 语 法

作为教师，
最大的危险就是只教词语而不教事物。
——Marc Block

引言——关键字——词法规则——程序——表达式——语句——声明——声明符——类
——派生类——特殊成员函数——重载——模板——异常处理——预处理命令

A.1 引言

这个有关C++语法的总结是为了帮助理解，它并不是对语言的一个准确陈述，特别是，这里所描述的语法接受的是合法C++程序的一个超集。必须使用排除歧义性的规则（A.5、A.7节）将表达式与声明区分开。进一步说，还需要使用访问控制、歧义性和类型规则等清除掉那些合乎语法但是却没有意义的结构。

这个C和C++标准文法采用语法的方式表述出每个细微的特征，而不是通过约束。这样做将得到精确性，但却未必总能提高可读性。

A.2 关键字

新的依赖于环境的关键字可以通过***typedef***（4.9.7节）、名字空间（8.2节）、类（第10章）、枚举（4.8节）和***template***（第13章）声明引进程序里。

```
typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-id

enum-name:
    identifier

template-name:
    identifier
```

注意，命名一个类的***typedef-name***本身也是一个***class-name***。

除非某个标识符被显式地声明为类型的名字，否则就假定它命名的是其他东西，而不是类型（见C.13.5节）。

C++关键字有：

C++关键字					
<i>and</i>	<i>and_eq</i>	<i>asm</i>	<i>auto</i>	<i>bitand</i>	<i>bitor</i>
<i>bool</i>	<i>break</i>	<i>case</i>	<i>catch</i>	<i>char</i>	<i>class</i>
<i>compl</i>	<i>const</i>	<i>const_cast</i>	<i>continue</i>	<i>default</i>	<i>delete</i>
<i>do</i>	<i>double</i>	<i>dynamic_cast</i>	<i>else</i>	<i>enum</i>	<i>explicit</i>
<i>export</i>	<i>extern</i>	<i>false</i>	<i>float</i>	<i>for</i>	<i>friend</i>
<i>goto</i>	<i>if</i>	<i>inline</i>	<i>int</i>	<i>long</i>	<i>mutable</i>
<i>namespace</i>	<i>new</i>	<i>not</i>	<i>not_eq</i>	<i>operator</i>	<i>or</i>
<i>or_eq</i>	<i>private</i>	<i>protected</i>	<i>public</i>	<i>register</i>	<i>reinterpret_cast</i>
<i>return</i>	<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>	<i>static_cast</i>
<i>struct</i>	<i>switch</i>	<i>template</i>	<i>this</i>	<i>throw</i>	<i>true</i>
<i>try</i>	<i>typedef</i>	<i>typeid</i>	<i>typename</i>	<i>union</i>	<i>unsigned</i>
<i>using</i>	<i>virtual</i>	<i>void</i>	<i>volatile</i>	<i>wchar_t</i>	<i>while</i>
<i>xor</i>	<i>xor_eq</i>				

A.3 词法规则

标准C和C++文法采用语法产生式给出词法规则。这样做能提高精确性，但也会形成一个比较大的文法，未必能增强可读性：

hex-quad:

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:

\u hex-quad

\U hex-quad hex-quad

preprocessing-token:

header-name

identifier

pp-number

character-literal

string-literal

preprocessing-op-or-punc

任何非空白的又不属于上述各项的字符

token:

identifier

keyword

literal

operator

punctuator

header-name:

<h-char-sequence>

"q-char-sequence"

h-char-sequence:

h-char

h-char-sequence h-char

h-char:

源字符集的任何成员, 除了换行符和 >

q-char-sequence:

q-char

q-char-sequence q-char

q-char:

源字符集的任何成员, 除了换行符和 "

pp-number:

digit

. digit

pp-number digit

pp-number nondigit

pp-number e sign

pp-number E sign

pp-number .

identifier:

nondigit

identifier nondigit

identifier digit

nondigit: 下述之一

universal-character-name

_ a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: 下述之一

0 1 2 3 4 5 6 7 8 9

preprocessing-op-or-punc: 下述之一

{	}	[]	#	##	()	<:	:>	<%	%>	%;%:
%:	;	:	?	::	.	.*	+	-	*	/	%	^
&		~	!	=	<	>	+=	-=	*=	/=	%=	^=
&=	=	<<=	>>=	<<	>>	==	!=	<=	>=	&&		++
--	,	->	->*	...	new	delete	and	and_eq	bitand			
bitor		compl	not	or	not_eq	xor	or_eq	xor_eq				

literal:

integer-literal

character-literal

floating-literal

string-literal

boolean-literal

integer-literal:

decimal-literal integer-suffix_{opt}

octal-literal integer-suffix_{opt}

hexadecimal-literal integer-suffix_{opt}

decimal-literal:

nonzero-digit

decimal-literal digit

octal-literal:

0

octal-literal octal-digit

hexadecimal-literal:

0x hexadecimal-digit

0X hexadecimal-digit

hexadecimal-literal hexadecimal-digit

nonzero-digit: 下述之一

1 2 3 4 5 6 7 8 9

octal-digit: one of

0 1 2 3 4 5 6 7

hexadecimal-digit: 下述之一

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:

unsigned-suffix *long-suffix*_{opt}

long-suffix *unsigned-suffix*_{opt}

unsigned-suffix: 下述之一

u U

long-suffix: 下述之一

l L

character-literal:

'c-char-sequence'

L'c-char-sequence'

c-char-sequence:

c-char

c-char-sequence c-char

c-char:

源字符集的任何成员, 除了单引号、反斜线和换行字符

escape-sequence

universal-character-name

escape-sequence:

simple-escape-sequence

octal-escape-sequence

hexadecimal-escape-sequence

simple-escape-sequence: 下述之一

\' \" \? \\ \a \b \f \n \r \t \v

octal-escape-sequence:

\ octal-digit

\ octal-digit octal-digit

\ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:

\x hexadecimal-digit

hexadecimal-escape-sequence hexadecimal-digit

floating-literal:

fractional-constant *exponent-part*_{opt} *floating-suffix*_{opt}

digit-sequence *exponent-part* *floating-suffix*_{opt}

fractional-constant:

*digit-sequence*_{opt} . *digit-sequence*

digit-sequence .

exponent-part:

e *sign*_{opt} *digit-sequence*

E *sign*_{opt} *digit-sequence*

sign: 下述之一

+ -

digit-sequence:
digit
digit-sequence digit

floating-suffix: 下述之一
f l F L

string-literal:
 " *s-char-sequence_{opt}* "
 L " *s-char-sequence_{opt}* "

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
 源字符集的任何成员, 除了双引号、反斜线和换行字符
escape-sequence
universal-character-name

boolean-literal:
false
true

A.4 程序

一个程序是一组*translation-unit*, 通过连接组合到一起(9.4节)。一个*translation-unit*通常被称为一个源文件, 它是一个*declaration*的序列:

translation-unit:
declaration-seq_{opt}

A.5 表达式

表达式在第6章介绍, 总结在6.2节。*expression-list*的定义与*expression*完全一样。存在两条规则来区分函数参数的逗号分隔符与逗号运算符(序列运算符)(6.2.2节)。

primary-expression:
literal
this
 :: *identifier*
 :: *operator-function-id*
 :: *qualified-id*
 (*expression*)
id-expression

id-expression:
unqualified-id
qualified-id

unqualified-id:
identifier
operator-function-id
conversion-function-id
 ~ *class-name*
template-id

qualified-id:
nested-name-specifier template_{opt} unqualified-id

nested-name-specifier:

class-or-namespace-name :: *nested-name-specifier*_{opt}
class-or-namespace-name :: *template nested-name-specifier*

class-or-namespace-name:

class-name
namespace-name

postfix-expression:

primary-expression
postfix-expression [*expression*]
postfix-expression { *expression-list*_{opt} }
simple-type-specifier { *expression-list*_{opt} }
typename ::_{opt} *nested-name-specifier identifier* (*expression-list*_{opt})
typename ::_{opt} *nested-name-specifier template*_{opt} *template-id* (*expression-list*_{opt})
postfix-expression . *template*_{opt} ::_{opt} *id-expression*
postfix-expression -> *template*_{opt} ::_{opt} *id-expression*
postfix-expression . *pseudo-destructor-name*
postfix-expression -> *pseudo-destructor-name*
postfix-expression ++
postfix-expression --
dynamic_cast < *type-id* > (*expression*)
static_cast < *type-id* > (*expression*)
reinterpret_cast < *type-id* > (*expression*)
const_cast < *type-id* > (*expression*)
typeid (*expression*)
typeid (*type-id*)

expression-list:

assignment-expression
expression-list , *assignment-expression*

pseudo-destructor-name:

::_{opt} *nested-name-specifier*_{opt} *type-name* :: ~ *type-name*
::_{opt} *nested-name-specifier template* *template-id* :: ~ *type-name*
::_{opt} *nested-name-specifier*_{opt} ~ *type-name*

unary-expression:

postfix-expression
++ *cast-expression*
-- *cast-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-id*)
new-expression
delete-expression

unary-operator: 下述之一

* & + - ! ~

new-expression:

::_{opt} *new new-placement*_{opt} *new-type-id new-initializer*_{opt}
::_{opt} *new new-placement*_{opt} (*type-id*) *new-initializer*_{opt}

new-placement:

(*expression-list*)

new-type-id:

*type-specifier-seq new-declarator*_{opt}

new-declarator:


```

ptr-operator new-declaratoropt
direct-new-declarator

direct-new-declarator:
    [ expression ]
    direct-new-declarator [ constant-expression ]

new-initializer:
    ( expression-listopt )

delete-expression:
    ::opt delete cast-expression
    ::opt delete [ ] cast-expression

cast-expression:
    unary-expression
    ( type-id ) cast-expression

pm-expression:
    cast-expression
    pm-expression . * cast-expression
    pm-expression -> * cast-expression

multiplicative-expression:
    pm-expression
    multiplicative-expression * pm-expression
    multiplicative-expression / pm-expression
    multiplicative-expression % pm-expression

additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression

shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression

relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression

and-expression:
    equality-expression
    and-expression & equality-expression

exclusive-or-expression:
    and-expression
    exclusive-or-expression ^ and-expression

inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression | exclusive-or-expression

```

logical-and-expression:
inclusive-or-expression
logical-and-expression && *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression ? *expression* : *assignment-expression*

assignment-expression:
conditional-expression
logical-or-expression *assignment-operator* *assignment-expression*
throw-expression

assignment-operator: 下述之一--
 = *= /= %+= -= >>= <<= &= ^= |=

expression:
assignment-expression
expression, *assignment-expression*

constant-expression:
conditional-expression

由于函数风格的强制和声明在形式上相似，这就产生了语法歧义性。例如，

```
int x;
void f()
{
    char(x); // 是把x转换为char，还是名为x的char变量声明？
}
```

所有这样的歧义性都将解析为声明。也就是说，“如果它可能被解析为声明，那么它就是一个声明。”例如：

```
T(a) -> m; // 表达式语句
T(a) ++; // 表达式语句
T(*e) (int(3)); // 声明
T(f) [4]; // 声明
T(a); // 声明
T(a) = m; // 声明
T(*b) (); // 声明
T(x), y, z = 7; // 声明
```

这种歧义性纯粹是语法上的，用于名字的仅有信息就是它是否已知为一个类型名或者一个模板名。如果这些都不能确定，那么就假定这个名字是命名了别的东西，而不是类型或者模板。

结构 *template unqualified-id* 用于在某个环境中说明 *unqualified-id* 是一个模板名，如果在那个环境里，无法通过推理得到这一结论（C.13.6节）。

A.6 语句

见6.3节。

statement:
labeled-statement

```

    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    declaration-statement
    try-block

labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement

expression-statement:
    expressionopt ;

compound-statement:
    { statement-seqopt }

statement-seq:
    statement
    statement-seq statement

selection-statement:
    if ( condition ) statement
    if ( condition ) statement else statement
    switch ( condition ) statement

condition:
    expression
    type-specifier-seq declarator = assignment-expression

iteration-statement:
    while ( condition ) statement
    do statement while ( expression ) ;
    for ( for-init-statement conditionopt ; expressionopt ) statement

for-init-statement:
    expression-statement
    simple-declaration

jump-statement:
    break ;
    continue ;
    return expressionopt ;
    goto identifier ;

declaration-statement:
    block-declaration

```

A.7 声明

声明的结构在第4章介绍，枚举在4.8节，指针和数组在第5章，函数在第7章，连接指示符在9.2.4节，存储类在10.4节。

```

declaration-seq:
    declaration
    declaration-seq declaration

declaration:
    block-declaration

```

- function-definition*
- template-declaration*
- explicit-instantiation*
- explicit-specialization*
- linkage-specification*
- namespace-definition*

block-declaration:

- simple-declaration*
- asm-definition*
- namespace-alias-definition*
- using-declaration*
- using-directive*

simple-declaration:

decl-specifier-seq_{opt} init-declarator-list_{opt} ;

decl-specifier:

- storage-class-specifier*
- type-specifier*
- function-specifier*
- friend*
- typedef*

decl-specifier-seq:

decl-specifier-seq_{opt} decl-specifier

storage-class-specifier:

- auto*
- register*
- static*
- extern*
- mutable*

function-specifier:

- inline*
- virtual*
- explicit*

typedef-name:

- identifier*

type-specifier:

- simple-type-specifier*
- class-specifier*
- enum-specifier*
- elaborated-type-specifier*
- cv-qualifier*

simple-type-specifier:

- ::_{opt} nested-name-specifier_{opt} type-name*
- ::_{opt} nested-name-specifier template_{opt} template-id*
- char*
- wchar_t*
- bool*
- short*
- int*
- long*
- signed*
- unsigned*
- float*

double
void

type-name:
class-name
enum-name
typedef-name

elaborated-type-specifier:
class-key : :_{opt} *nested-name-specifier*_{opt} *identifier*
enum : :_{opt} *nested-name-specifier*_{opt} *identifier*
typename : :_{opt} *nested-name-specifier* *identifier*
typename : :_{opt} *nested-name-specifier* *template*_{opt} *template-id*

enum-name:
identifier

enum-specifier:
enum *identifier*_{opt} { *enumerator-list*_{opt} }

enumerator-list:
enumerator-definition
enumerator-list , *enumerator-definition*

enumerator-definition:
enumerator
enumerator = *constant-expression*

enumerator:
identifier

namespace-name:
original-namespace-name
namespace-alias

original-namespace-name:
identifier

namespace-definition:
named-namespace-definition
unnamed-namespace-definition

named-namespace-definition:
original-namespace-definition
extension-namespace-definition

original-namespace-definition:
namespace *identifier* { *namespace-body* }

extension-namespace-definition:
namespace *original-namespace-name* { *namespace-body* }

unnamed-namespace-definition:
namespace { *namespace-body* }

namespace-body:
*declaration-seq*_{opt}

namespace-alias:
identifier

namespace-alias-definition:
namespace *identifier* = *qualified-namespace-specifier* ;

qualified-namespace-specifier:

```

::opt nested-name-specifieropt namespace-name

using-declaration:
    using typenameopt ::opt nested-name-specifier unqualified-id ;
    using :: unqualified-id ;

using-directive:
    using namespace ::opt nested-name-specifieropt namespace-name ;

asm-definition:
    asm ( string-literal ) ;

linkage-specification:
    extern string-literal { declaration-seqopt }
    extern string-literal declaration

```

语法允许声明的任意嵌套，但对此也存在某些语义限制。例如，不允许嵌套的函数（不能将函数定义为其他函数局部的）。

声明开始处的描述符列表不能为空（不存在“隐含的`int`”，B.2节），这个表由最长的可能描述符序列构成。例如，

```

typedef int I;
void f(unsigned I) { /* ... */ }

```

在这里，`f()` 以一个没给出名字的`unsigned int`为参数。

一个`asm()` 是一段汇编代码插入，其意义由实现定义，其意图就是要求一个表示汇编代码段的字符序列，将这段代码插入程序所生成代码里的这个指定位置。

将变量声明为`register`是一个给编译程序的提示，希望做针对频繁访问的优化。对于大部分编译器而言，这样做都是多余的。

A.7.1 声明符

见4.9.1节、第5章（指针和数组）、7.7节（指向函数的指针）和15.5节（指向成员的指针）。

```

init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator

init-declarator:
    declarator initializeropt

declarator:
    direct-declarator
    ptr-operator declarator

direct-declarator:
    declarator-id
    direct-declarator ( parameter-declaration-clause ) cv-qualifier-seqopt exception-specificationopt
    direct-declarator [ constant-expressionopt ]
    ( declarator )

ptr-operator:
    * cv-qualifier-seqopt
    &
    ::opt nested-name-specifier * cv-qualifier-seqopt

cv-qualifier-seq:
    cv-qualifier cv-qualifier-seqopt

```

```

cv-qualifier:
    const
    volatile

declarator-id:
    ::opt id-expression
    ::opt nested-name-specifieropt type-name

type-id:
    type-specifier-seq abstract-declaratoropt

type-specifier-seq:
    type-specifier type-specifier-seqopt

abstract-declarator:
    ptr-operator abstract-declaratoropt
    direct-abstract-declarator

direct-abstract-declarator:
    direct-abstract-declaratoropt ( parameter-declaration-clause ) cv-qualifier-seqopt exception-specificationopt
    direct-abstract-declaratoropt [ constant-expressionopt ]
    ( abstract-declarator )

parameter-declaration-clause:
    parameter-declaration-listopt ...opt
    parameter-declaration-list , ...

parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list , parameter-declaration

parameter-declaration:
    decl-specifier-seq declarator
    decl-specifier-seq declarator = assignment-expression
    decl-specifier-seq abstract-declaratoropt
    decl-specifier-seq abstract-declaratoropt = assignment-expression

function-definition:
    decl-specifier-seqopt declarator ctor-initializeropt function-body
    decl-specifier-seqopt declarator function-try-block

function-body:
    compound-statement

initializer:
    = initializer-clause
    ( expression-list )

initializer-clause:
    assignment-expression
    { initializer-list ,opt }
    { }

initializer-list:
    initializer-clause
    initializer-list , initializer-clause

```

volatile描述符是给编译程序的一个提示，说明这个变量的值可能以语言未描述的方式改变，因此必须避免去做过于激进的优化。例如，一个实时时钟可能声明如下：

```
extern const volatile long clock;
```

对**clock**的任意两次读操作都可能得到不同结果。

A.8 类

见第10章。

class-name:

identifier
template-id

class-specifier:

class-head { *member-specification*_{opt} }

class-head:

class-key *identifier*_{opt} *base-clause*_{opt}
class-key *nested-name-specifier* *identifier* *base-clause*_{opt}
class-key *nested-name-specifier* *template* *template-id* *base-clause*_{opt}

class-key:

class
struct
union

member-specification:

member-declaration *member-specification*_{opt}
access-specifier : *member-specification*_{opt}

member-declaration:

*decl-specifier-seq*_{opt} *member-declarator-list*_{opt} ;
function-definition ;_{opt}
: :_{opt} *nested-name-specifier* *template*_{opt} *unqualified-id* ;
using-declaration
template-declaration

member-declarator-list:

member-declarator
member-declarator-list , *member-declarator*

member-declarator:

declarator *pure-specifier*_{opt}
declarator *constant-initializer*_{opt}
*identifier*_{opt} : *constant-expression*

pure-specifier:

= 0

constant-initializer:

= *constant-expression*

为保持与C的兼容性，在同一个作用域里，可以声明一个类和一个具有同样名字的非类成分。例如，

```
struct stat { /* ... */ };
int stat(char* name, struct stat* buf);
```

在这种情况下，普通的名字（*stat*）就是那个非类的成分的名字，而在类名之前就必须使用 *class-key* 作为前缀。

常量表达式（*constant expression*）在C.5节定义。

A.8.1 派生类

见第12章和第15章。


```

base-clause:
    : base-specifier-list

base-specifier-list:
    base-specifier
    base-specifier-list , base-specifier

base-specifier:
    :opt nested-name-specifieropt class-name
    virtual access-specifieropt :opt nested-name-specifieropt class-name
    access-specifier virtualopt :opt nested-name-specifieropt class-name

access-specifier:
    private
    protected
    public

```

A.8.2 特殊成员函数

见11.4节（转换运算符）、10.4.6节（类成员初始化）和12.2.2节（基类初始化）。

```

conversion-function-id:
    operator conversion-type-id

conversion-type-id:
    type-specifier-seq conversion-declaratoropt

conversion-declarator:
    ptr-operator conversion-declaratoropt

ctor-initializer:
    : mem-initializer-list

mem-initializer-list:
    mem-initializer
    mem-initializer , mem-initializer-list

mem-initializer:
    mem-initializer-id ( expression-listopt )

mem-initializer-id:
    :opt nested-name-specifieropt class-name
    identifier

```

A.8.3 重载

见第11章。

```

operator-function-id:
    operator operator

operator: 下述之一
    new delete new[] delete[]
    + - * / % ^ & | ~ ! = < >
    += -= *= /= %= ^= &= |= << >> >>= <<= ==
    != <= >= && || ++ -- , ->* -> () []

```

A.9 模板

模板在第13章和C.13节解释。

```

template-declaration:
    exportopt template < template-parameter-list > declaration

template-parameter-list:
    template-parameter
    template-parameter-list , template-parameter

template-parameter:
    type-parameter
    parameter-declaration

type-parameter:
    class identifieropt
    class identifieropt = type-id
    typename identifieropt
    typename identifieropt = type-id
    template < template-parameter-list > class identifieropt
    template < template-parameter-list > class identifieropt = template-name

template-id:
    template-name < template-argument-listopt >

template-name:
    identifier

template-argument-list:
    template-argument
    template-argument-list , template-argument

template-argument:
    assignment-expression
    type-id
    template-name

explicit-instantiation:
    template declaration

explicit-specialization:
    template < > declaration

```

显式的模板参数专门化打开了一种可能的很不清楚的语法歧义性。考虑

```

void h()
{
    f<1>(0); // 歧义: ((f) < 1) > 0 或者 (f<1>)(0)?
             // 解析: 用参数0调用f<1>
}

```

有关的解析简单而有效: 如果 f 为模板名, $f<$ 就是一个限定了的模板名的开始, 随后的词法单词必须基于这一点进行解释; 如果不是, $<$ 就表示小于。与此类似, 第一个非嵌套的 $>$ 结束这个模板参数列表。如果嵌套中有大于符号, 那么就必须加括号:

```

f< a>b > (0); // 语法错
f< (a>b) > (0); // ok

```

如果两个 $>$ 过于靠近, 就可能出现另一种类似的语法歧义性。例如,

```

list<vector<int>>> lv1; // 语法错: 未预期的>> (右移)
list< vector<int> > lv2; // 正确: 向量的表

```

注意两个 $>$ 之间的空格, $>>$ 是右移运算符。这确实很讨厌。

A.10 异常处理

见8.3节和第14章。

```

try-block:
    try compound-statement handler-seq

function-try-block:
    try ctor-initializeropt function-body handler-seq

handler-seq:
    handler handler-seqopt

handler:
    catch ( exception-declaration ) compound-statement

exception-declaration:
    type-specifier-seq declarator
    type-specifier-seq abstract-declarator
    type-specifier-seq
    ...

throw-expression:
    throw assignment-expressionopt

exception-specification:
    throw ( type-id-listopt )

type-id-list:
    type-id
    type-id-list , type-id
  
```

A.11 预处理指令

预处理器是一个相对而言不太复杂的宏处理程序，它基本上是在词法单词的基础上工作的（而不是基于单个的字符）。除了定义和使用宏的能力（7.8节）之外，预处理器还提供了一些机制，用于包含正文文件和标准头文件（9.2.1节）以及基于宏的条件编译（9.3.3节）。例如，

```

#if OPT==4
#include "header4.h"
#elif 0<OPT
#include "someheader.h"
#else
#include<cstdlib>
#endif
  
```

所有的预处理指令都以 # 开始，这个符号必须是所在行里第一个非空白字符：

```

preprocessing-file:
    groupopt

group:
    group-part
    group group-part

group-part:
    pp-tokensopt new-line
    if-section
    control-line
  
```

if-section:

if-group elif-groups_{opt} else-group_{opt} endif-line

if-group:

if constant-expression new-line group_{opt}

ifdef identifier new-line group_{opt}

ifndef identifier new-line group_{opt}

elif-groups:

elif-group

elif-groups elif-group

elif-group:

elif constant-expression new-line group_{opt}

else-group:

else new-line group_{opt}

endif-line:

endif new-line

control-line:

include pp-tokens new-line

define identifier replacement-list new-line

define identifier lparen identifier-list_{opt}) replacement-list new-line

undef identifier new-line

line pp-tokens new-line

error pp-tokens_{opt} new-line

pragma pp-tokens_{opt} new-line

new-line

lparen:

左圆括号字符，在它之前不能有 *white-space*

replacement-list:

pp-tokens_{opt}

pp-tokens:

preprocessing-token

pp-tokens preprocessing-token

new-line:

the new-line character

identifier-list:

identifier

identifier-list , identifier

附录B 兼容性

你按你的习惯继续吧，
而我按我的习惯
——C. Napier

C/C++兼容性——C和C++之间无声的差异——不是C++的C代码——贬斥的特征——不是C的C++代码——对付老的C++实现——头文件——标准库——名字空间——分配错误——模板——for语句初始化表达式——忠告——练习

B.1 引言

本附录描述在C和C++、标准C++和C++的早期版本之间的不兼容性，其目的是列出所有可能给程序员造成问题的差异，并指出处理这些问题的一些方式。在人们企图将C程序修改为C++程序时，企图将C++程序从C++的标准前的一个版本移植到另一个版本时，或者企图用某个老的编译器编译使用到新C++特征的程序时，都可能发现兼容性的问题。这里的目标并不是给你勾画出在某个实现中可能出现的每一个兼容性问题的细节，而是列出最可能出现的问题，并介绍解决它们的标准方法。

当你考察有关兼容性的情况时，一个最关键的问题就是考虑一下程序应当能在哪些实现上工作。为了学习C++，使用最完整和最有帮助的实现是很有意义的，而为了发布一个产品，更谨慎的策略可能是使产品能运行在尽可能多的系统上。在过去，这一点也是人们避免C++新特征的理由（有时不过是口实）。无论如何，实现正在向标准收敛，所以，有关跨平台兼容性的需要已经不像几年前那样成为严重关注的问题了。

B.2 C/C++兼容性

除了一些次要的例外，C++是C的一个超集。大部分差异的根源是C++更多地强调了类型检查。书写良好的C程序倾向于同时也是C++程序。编译器可以诊断出在C++和C之间的所有差异。

B.2.1 “无声的”差异

除了极少的例外，所有同时是C++和C的程序在两种语言中将具有同样的意义。幸运的是，这些“无声的”差异相当少见。

在C里，字符常量和枚举符的大小都是`sizeof(int)`；而在C++里，`sizeof('a')`等于`sizeof(char)`，且C++实现可以自己为枚举选择最合适的大小（4.8节）。

C++提供了//注释，而C没有（虽然许多C实现提供了这种注释形式作为扩充）。这一差异可能用于构造出在两种语言里意义不同的程序。例如，

```
int f(int a, int b)
{
    return a // * 不大可能 */ b
    ;    /* 不实际; 分号放在另一行以避免语法错误 */
}
```

ISO C正在修订, 以便像C++这样允许//。

在内层作用域里声明的结构名将屏蔽位于外层作用域的对象、函数、枚举或者类型的名字。例如:

```
int x[99];
void f()
{
    struct x { int a; };
    sizeof(x); /* 在C中是数组x的大小, 在C++中是结构x的大小 */
}
```

B.2.2 不是C++的C代码

C/C++之间的不兼容所引起的大部分问题都不难捉摸, 编译器很容易捕捉到其中大部分问题。本节将给出一些不是C++的C代码实例。其中大多数都被认为具有极糟的风格, 甚至是现代C中过时的东西。

在C里, 大部分函数可以在没有预先声明的情况下调用。例如,

```
main()    /* 糟糕风格的C, 不是C++ */
{
    double sq2 = sqrt(2);                /* 调用未声明的函数 */
    printf("the square root of 2 is %g\n", sq2); /* 调用未声明的函数 */
}
```

一般说, C语言建议完全而一致地使用函数声明 (函数原型)。在那些遵循这种明智建议的地方, 特别是在C编译器提供了选项要求遵循这些建议的地方, C代码就会符合C++规则。在调用未声明的函数时, 你必须对函数和C规则都非常熟悉, 知道你是否造成错误或者引入了兼容性问题。举例来说, 作为C程序, 前面这个main()里至少有两个错误。

在C里, 对于那些声明中没有描述参数类型的函数, 它们就可以取任意个数的任意类型的参数。这种使用方式在标准C里被作为过时形式, 但却相当常见:

```
void f(); /* 未提到参数类型 */
void g()
{
    f(2);    /* 糟糕风格的C, 不是C++ */
}
```

在C语言里, 函数定义可以采用另一种语法, 在参数表之后另外描述参数的类型:

```
void f(a,p,c) char *p; char c; { /* ... */ } /* C程序, 不是C++ */
```

这个定义必须重写为:

```
void f(int a, char* p, char c) { /* ... */ }
```

在C和C++标准以前的版本中, 类型描述符默认为int。例如,

```
const a = 7; /* C中假定类型为int, 不是C++ */
```

ISO C正在修订, 将像C++一样不允许“隐含的int”。

C允许将**struct**定义用在返回值类型和参数类型声明中。例如，

```
struct S { int x,y; } f();          /* C程序，不是C++ */
void g(struct S { int x,y; } y);    /* C程序，不是C++ */
```

C++的类型定义规则使这种声明不再有意义，因此也不再允许它。

在C里可用将整数赋给枚举类型的变量：

```
enum Direction { up, down };
enum Direction d = 1;              /* 错误：用int给Direction赋值。C语言中允许
```

C++提供的关键字比C多得多，如果这些关键字中的某个被作为C程序里的标识符，这个程序就必须经过修改才能成为C++程序：

不是C关键字的C++关键字					
<i>and</i>	<i>and_eq</i>	<i>asm</i>	<i>bitand</i>	<i>bitor</i>	<i>bool</i>
<i>catch</i>	<i>class</i>	<i>compl</i>	<i>const_cast</i>	<i>delete</i>	<i>dynamic_cast</i>
<i>explicit</i>	<i>export</i>	<i>false</i>	<i>friend</i>	<i>inline</i>	<i>mutable</i>
<i>namespace</i>	<i>new</i>	<i>not</i>	<i>not_eq</i>	<i>operator</i>	<i>or</i>
<i>or_eq</i>	<i>private</i>	<i>protected</i>	<i>public</i>	<i>reinterpret_cast</i>	<i>static_cast</i>
<i>template</i>	<i>this</i>	<i>throw</i>	<i>true</i>	<i>try</i>	<i>typeid</i>
<i>typename</i>	<i>using</i>	<i>virtual</i>	<i>wchar_t</i>	<i>xor</i>	<i>xor_eq</i>

在C里，某些C++关键字被作为宏定义在头文件里：

被作为C宏的C++关键字					
<i>and</i>	<i>and_eq</i>	<i>bitand</i>	<i>bitor</i>	<i>compl</i>	<i>not</i>
<i>not_eq</i>	<i>or</i>	<i>or_eq</i>	<i>wchar_t</i>	<i>xor</i>	<i>xor_eq</i>

这也意味着它们可以在C里用 **#ifdef** 检测、重新定义等等。

在C里，一个全局数据对象可以在同一个编译单位里声明多次，不必使用**extern**描述符，条件是在这些声明中至多只有一个提供了初始化表达式，在这种情况下就认为对象只定义了一次。例如：

```
int i; int i;    /* 定义或声明了一个整数i，不是C++ */
```

在C++里，每个实体必须定义恰好一次；9.2.3节。

在C++里，在一个作用域里，不能有一个类名字与一个为引用其他类型而声明的**typedef**的名字相同；5.7节。

在C里，**void*** 可以作为对任意指针类型的变量赋值操作右边的运算对象，或用于对这种变量的初始化；在C++里不能这样做（5.6节）。例如，

```
void f(int n)
{
    int* p = malloc(n*sizeof(int)); /* 不是C++。在C++里，分配用new */
}
```

C允许跳过初始化，而C++不允许。

在C里全局**const**默认地具有外部连接；在C++里除非明确声明为**extern**，否则就不是这样，而且它们必须显式地初始化（5.4节）。

在C里，嵌套结构的名称与它们嵌套于其中的结构放在同一个作用域里。例如，

```

struct S {
    struct T { /* ... */ };
    // ...
};

struct T x;    /* 在C里可以，意思是“S::T x;”，不是C++ */

```

在C里，数组可以用多于它所需元素个数的初始式进行初始化。例如：

```
char v[5] = "Oscar";    /* 在C里可以，不使用结束的0。不是C++ */
```

B.2.3 贬斥的特征

通过贬斥某个特征，标准化委员会表达了一种期望：这个特征应该靠边站。当然，标准化委员会并没有权力去删除某个广泛使用的特征——无论它是多么多余或者危险。这样，所谓贬斥就是一个很强的提示，希望用户避免去用它。

关键字`static`原意是“静态分配”，但也可以用于指明一个函数或者对象是某编译单位中局部的东西。例如，

```

// file1:
static int glob;

// file2:
static int glob;

```

这个程序实际上有两个称为`glob`的整数，每个整数由它所在的编译单位里的函数排它地使用。

C++贬斥这种用`static`表示“局部于编译单位”的用法，用无名名字空间（8.2.5.1节）替代。

隐式地将字符串文字量转换到（非`const`）`char*`也受到贬斥。应该用有名字的`char`数组，或者避免用字符串文字量给`char*`赋值（5.2.2节）。

在引进了新风格的强制以后，就应该贬斥C风格的强制。程序员应该特别认真地考虑在自己的程序里禁止使用C风格的强制。在所有需要类型转换的地方，`static_cast`、`reinterpret_cast`、`const_cast`或者它们的组合能够完成C风格强制可以做的所有事情。偏向新风格强制是因为它们更明显也更容易看清楚（6.2.7节）。

B.2.4 不是C的C++代码

本节列出C++所提供而C没有提供的功能，这些特征按照用途排序。当然，还可以有许多分类方式，大部分特征能够服务于多种用途，所以不要把这里的分类方式看得太认真。

— 主要是为了记述方便的特征：

- [1] // 注释（2.3节）；正被加入C。
- [2] 支持受限的字符集（C.3.1节）。
- [3] 支持扩充的字符集（C.3.3节）；正被加入C。
- [4] 对`static`存储中对象的非常量初始式（9.4.1节）。
- [5] 常量表达式中的`const`（5.4节、C.5节）。
- [6] 将声明作为语句（6.3.1节）。
- [7] 在for语句初始化表达式和条件中的声明（6.3.3节、6.3.2.1节）。
- [8] 结构名无须前缀`struct`（5.7节）。

— 主要是为了增强类型系统的特征：

- [1] 函数参数的类型检查 (7.1 节); 后来被加入C (B.2.2 节)。
- [2] 类型安全的连接 (9.2 节、9.2.3 节)。
- [3] 用`new`和`delete`管理自由存储 (6.2.6 节、10.4.5 节、15.6 节)。
- [4] `const` (5.4、5.4.1 节); 后来被加入C。
- [5] 布尔类型`bool` (4.2 节)。
- [6] 新的强制语法 (6.2.7 节)。

— 服务于用户定义类型的功能:

- [1] 类 (第10章)。
- [2] 成员函数 (10.2.1 节) 和成员类 (11.12 节)。
- [3] 构造函数和析构函数 (10.2.3 节、10.4.1 节)。
- [4] 派生类 (第12章、第15章)。
- [5] `virtual`函数和抽象类 (12.2.6 节、12.3 节)。
- [6] 公用/保护/私用访问控制 (10.2.2 节、15.3 节、C.11 节)。
- [7] `friend` (11.5 节)。
- [8] 指向成员的指针 (15.5 节、C.12 节)。
- [9] `static`成员 (10.2.4 节)。
- [10] `mutable`成员 (10.2.7.2 节)。
- [11] 运算符重载 (第11章)。
- [12] 引用 (5.5 节)。

— 主要是为了程序组织的特征 (除了类之外的其他特征):

- [1] 模板 (第13章、C.13 节)。
- [2] 内联函数 (7.1.1 节)。
- [3] 默认参数 (7.5 节)。
- [4] 函数重载 (7.4 节)。
- [5] 名字空间 (8.2 节)。
- [6] 显式作用域限定 (运算符`::`, 4.9.4 节)。
- [7] 异常处理 (8.3 节、第14章)。
- [8] 运行时类型识别 (15.4 节)。

C++所增加的关键字 (B.2.2 节) 可以用于标明大部分C++的特殊功能。当然也有些功能 (例如函数重载和常量表达式里的`const`) 并没有特定关键字标明。除了这里列出的语言特征外, C++标准库 (16.1.2 节) 中的大部分都是C++特殊的东西。

宏 `__cplusplus` 可以用于确定是C还是C++编译器正在处理这个程序 (9.2.4 节)。

B.3 对付老的C++实现

从1983年起C++就一直在使用。从那以来已经定义过几个版本, 也出现了许多分别开发出的实现。标准化努力的基本目标是保证实现者和使用者能够有一个统一的C++定义, 由此出发开始各自的工作。然而, 在这个定义浸透了整个C++社会之前, 我们将不得不处理这样的事实: 并不是每个C++实现都提供了本书所描述的每一个C++特征。

在人们第一次认真地查看C++时用的却是某个5年之前的实现, 这种情况很不幸, 但也是

常见的。一个典型的原因是这种实现广泛可用，而且不要钱。如果让人选择，任何有自尊的专业人员都不会愿意去用这样的古董。对于新手来说，较老的实现则会带来沉重的隐蔽代价。缺乏许多语言特征和库的支持，就意味着这些新手必须去与那些早已被新的实现清除的问题战斗。使用特征不足的老实现也会束缚这些新人的程序设计风格，并给他们对于C++是什么的认识造成偏见。开始学习C++的最好子集并不是那组低级特征（也不是C与C++的公共子集；1.2节）。我特别建议依赖于标准库和模板，以使学习更容易，也将对能够怎样去做C++程序设计取得一个正确的最初印象。

C++的第一个商业版本是在1985年，其语言由本书的第一版定义。在那个时候，C++还没有多重继承、模板、运行时类型信息、异常和名字空间。今天，如果一个实现没有提供这些特征中至少几个的话，我看不到任何去用它的理由。我在1989年将多重继承、模板和异常加入了C++的定义。不过，早期实现对模板和异常的支持参差不齐，常常很糟糕。如果你在某个老实现中用模板或者异常遇到了问题，请考虑立即升级系统。

总而言之，最明智的是去用一个在所有可能之处都符合标准的实现，并尽可能减少对由实现定义的和语言中未定义方面的依赖性。设计时应该就像完整的语言都可以用，而后在需要时做一些迂回工作。这样做，与为C++的最小公因子设计相比，得到的程序将组织得更好也更容易维护。还有，请谨慎使用具体实现的特殊语言扩充，只在绝对需要时才去用它们。

B.3.1 头文件

传统上每个头文件都以 *.h* 作为后缀。C++实现也提供了如 *<map.h>* 和 *<iostream.h>* 这样的头文件。为了兼容性，大部分实现都是这样。

当标准化委员会需要一些头文件，以便重新定义标准库的版本和新的库功能时，如何为这些头文件命名就变成了一个问题。采用老的 *.h* 名字将导致兼容性问题，最后的解决方法就是去掉标准头文件名字的 *.h* 后缀。这个后缀原本就是多余的，因为 *<* 和 *>* 记法本身已经指明这是标准头文件的名字。

这样，标准库提供了一组不带后缀的头文件，比如说 *<iostream>* 和 *<map>*。这些文件里的声明都放入名字空间 *std*。老的头文件将声明放入全局名字空间，并采用 *.h* 后缀。例如，

```
#include<iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

如果在某个实现中这一程序无法编译，请试试更传统的形式：

```
#include<iostream.h>

int main()
{
    cout << "Hello, world!\n";
}
```

一些最严重的移植问题就是由于不兼容的头文件引起的。标准头文件对此只能有很小的贡献，因为一个程序常常依赖于一大批并非每个系统都有的头文件，或依赖于一大批并非在每个系统里都出现在同样头文件里的声明，或依赖于某些看起来是标准的声明（因为它们出现在具有标准名字的头文件里），而实际上这些声明并不是标准的一部分。

在面对相互冲突的头文件时，不存在任何完全令人满意的可移植性解决方案。一种普遍的想法是避免直接依赖于不一致的头文件，并将剩下的依赖关系局部化。也就是说，我们试图通过间接和局部化来达到可移植性。举例来说，如果我们所需的声明在不同系统里由不同的头文件给出，我们可以选择 `#include` 具体应用的一个特殊头文件，转由它去 `#include` 各个系统里的适当头文件。与此类似，如果不同系统中提供的某些功能在形式上有些差异，我们也可以选择采用专用的界面类或者函数去访问这些功能。

B.3.2 标准库

很自然，标准之前的C++实现里可能缺少标准库的某些部分。大部分应该有 `iostream`，非模板的 `complex`，一个有些不同的 `string` 类，以及C的标准库，但有些可能缺少 `map`、`list` 和 `valarray` 等。在这些情况下，可以采取某种方式先用可用的（常常是专有的）库，这种方式应允许你在实现升级到标准之后转过去。在缺少标准库类时，使用非标准的 `string`、`list` 或 `map` 也比回到C风格的程序设计好一些。还有，存在标准库的STL部分（第16、17、18、19章）的很好的实现，可以自由下载。

标准库的早期实现是不完全的。例如，某些可能带有不支持分配器的容器，另一些则要求对每个类都显式地描述分配器。类似问题也出现在其他“策略参数”方面，如比较准则等。下面是一些例子：

```
list<int> li; // 可以，但某些实现要求一个分配器
list<int, allocator<int>> li2; // 可以，但某些实现里未实现分配器

map<string, Record> m1; // 可以，但某些实现要求有小于操作
map<string, Record, less<string>> m2;
```

请采用实现能够接受的方式。这些实现最终将接受所有的东西。

较早的C++实现提供了在 `<strstream.h>` 里定义的 `istrstream` 和 `ostrstream`，而没有提供在 `<sstream>` 中定义的 `istringstream` 和 `ostingstream`。这些 `strstream` 直接在 `char[]` 上操作（见 21.10[26]）。

在标准之前的C++实现里的流没有参数化。特别是带有 `basic_` 前缀的模板是标准里的新东西，`basic_ios` 类过去叫做 `ios`。有趣的是，`iosstate` 过去被叫作 `io_state`。

B.3.3 名字空间

如果你所用的实现不支持名字空间，那么请用源文件来表示程序的逻辑结构（第9章）。类似地，用头文件表示你为实现提供的界面，或者与C共享的界面。

在缺少名字空间的情况下，可以利用 `static` 来缓和由于缺乏名字空间而产生的问题。也可以给全局名字共同的前缀，以区分你的名字与其他部分的代码里的名字。例如，

```
// 用于无名字空间的实现
class bs_string { /* ... */ }; // Bjarne的串
typedef int bs_bool; // Bjarne的布尔类型

class joe_string; // Joe的串
enum joe_bool { joe_false, joe_true }; // Joe的布尔类型
```

选择前缀时请当心，现存的C和C++库里到处都是这类前缀。

B.3.4 分配错误

在异常处理之前的C++里，运算符`new`返回0以指明分配失败，标准C++的`new`按照默认方式是抛出`bad_alloc`。

一般而论，最好是转到标准方面。在这一情况下，这样做意味着要修改许多代码去捕捉`bad_alloc`而不是检测0。在这两种情况下，要对付存储耗尽问题，而不只是给出一个错误信息，在许多系统里都非常困难。

然而，在那些无法实际地将检测0修改为捕捉`bad_alloc`的地方，你可能需要在某些时候修改程序，使之回到异常处理行为之前的情况。如果没有安装`_new_handler`，使用`nothrow`分配符就能导致在出现分配失误时返回0：

```
X* p1 = new X;           // 没有存储时抛出bad_alloc
X* p2 = new(nothrow) X;  // 没有存储时返回0
```

B.3.5 模板

标准引入了新的模板特征，并澄清了若干现存的规则。

如果你的实现不支持部分专门化，请为那个原本可以作为专门化的模板使用另一个名字。例如，

```
template<class T> class plist : private list<void*> { // 本应是list<T*>
    // ...
};
```

如果你的实现不支持成员模板，有些技术就行不通了。特别是，成员模板使程序员可以描述具有特殊灵活性的构造和转换，没有它就达不到这种灵活性（13.6.2节）。有时可以提供一个非成员函数，以它作为创建对象的替代方式。考虑

```
template<class T> class X {
    // ...
    template<class A> X(const A& a);
};
```

如果缺少成员模板，我们就必须将自己限制到特定类型：

```
template<class T> class X {
    // ...
    X(const A1& a);
    X(const A2& a);
    // ...
};
```

当一个模板类实例化时，许多早些的实现将为该模板类中定义的所有成员函数生成定义。这可能在没有使用的函数里引起错误（C.13.9.1节），解决方法是将成员函数的定义放到类声明的后面。例如，不要写

```
template<class T> class Container {
    // ...
public:
    void sort() { /* 用< */ } // 在类中的定义
};

class Glob { /* 对Glob无< */ };

Container<Glob> cg; // 某些标准前的实现将试图定义Container<Glob>::sort()
```

而用

```
template<class T> class Container {
    // ...
public:
    void sort();
};

template<class T> void Container<T>::sort() { /* use < */ } // 在类外的定义

class Glob { /* 对Glob无< */ };

Container<Glob> cg; // 只要不调用cg.sort()就不会有问题
```

C++的早期实现不能处理类中成员定义在后的情况。例如,

```
template<class T> class Vector {
public:
    T& operator[] (size_t i) { return v[i]; } // v在下面声明
    // ...
private:
    T* v; // 呜呼:找不到!
    size_t sz;
};
```

在这种情况下,或者是重排成员以避免问题,或者是将成员函数的定义放在类声明之后。

有些标准之前的C++实现不接受模板的默认参数(13.4.1节)。在这种情况下,对每个模板形式参数都必须给出显式参数。例如,

```
template<class Key, class T, class LT = less<T> > class map {
    // ...
};

map<string,int> m; // 呜呼:未实现默认模板参数
map< string,int, less<string> > m2; // 迂回工作:显式给出
```

B.3.6 for语句的初始化表达式

考虑

```
void f(vector<char>& v, int m)
{
    for (int i=0; i<v.size() && i<=m; ++i) cout << v[i];
    if (i == m) { // 错误:在for语句结束后引用i
        // ...
    }
}
```

这样的代码也可能工作,因为按照C++原来的定义,这种控制变量的作用域一直延续到for语句出现的那个作用域的结束。如果你看到这种代码,那就简单地将控制变量的声明移到for语句之前:

```
void f2(vector<char>& v, int m)
{
    int i=0; // 循环后需要的i
    for (; i<v.size() && i<=m; ++i) cout << v[i];
    if (i == m) {
        // ...
    }
}
```

```
    }
}
```

B.4 忠告

- [1] 要学习C++, 应该使用你可以得到的标准C++的最新的和完全的实现; B.3节。
- [2] C和C++的公共子集并不是学习C++时最好的开始子集; 1.6节、B.3节。
- [3] 对于产品代码, 请记住并不是每个C++实现都是完全的最新的。在产品代码中使用某个新特征之前应先做试验, 写一个小程序, 测试你计划使用的实现与标准的相符情况和性能。例如, 参见8.5[6~7]、16.5[10]和B.5[7]。
- [4] 避免被贬斥的特征, 例如全局的`static`; 还应避免C风格的强制; 6.2.7节、B.2.3节。
- [5] “隐含的`int`”已禁止, 因此请明确描述每个函数、变量、`const`等的类型; B.2.2节。
- [6] 在将C程序转为C++程序时, 首先保证函数声明(原型)和标准头文件的一致使用; B.2.2节。
- [7] 在将C程序转为C++程序时, 对以C++关键字为名的变量重新命名; B.2.2节。
- [8] 在将C程序转为C++程序时, 将`malloc()`的结果强制到适当类型, 或者将`malloc()`的所有使用都改为`new`; B.2.2节。
- [9] 在将`malloc()`和`free()`转为`new`和`delete`时, 请考虑用`vector`、`push_back()`和`reserve()`而不是`realloc()`; 3.8节、16.3.5节。
- [10] 在将C程序转为C++程序时, 记住这里没有从`int`到枚举的隐式转换; 如果需要, 请用显式转换; 4.8节。
- [11] 在名字空间`std`里定义的功能都定义在无后缀的头文件里(例如, `std::cout`声明在`<iostream>`里)。早些的实现将标准库功能定义在全局空间里, 声明在带`.h`后缀的头文件里(例如, `std::cout`声明在`<iostream.h>`里); 9.2.2节、B.3.1节。
- [12] 如果老的代码检测`new`的结果是否为0, 那么必须将它修改为捕捉`bad_alloc`或者使用`new(nothrow)`; B.3.4节。
- [13] 如果你用的实现不支持默认模板参数, 请显式提供参数; 用`typedef`可以避免重复写模板参数(类似于`string`的`typedef`使你无须写`basic_string<char, char_traits<char>, allocator<char>>`); B.3.5节。
- [14] 用`<string>`得到`std::string`(`<string.h>`里保存的是C风格的串函数); 9.2.2节、B.3.1节。
- [15] 对每个标准C头文件`<X.h>`, 它将名字放入全局名字空间; 与之对应的头文件`<cX>`将名字放入名字空间`std`; B.3.1节。
- [16] 许多系统有一个“`String.h`”头文件里定义了一个串类型。注意, 这个串类型与标准库的`string`不同。
- [17] 尽可能使用标准库功能, 而不是非标准的功能; 20.1节、B.3节、C.2节。
- [18] 在声明C函数时用`extern "C"`; 9.2.4节

B.5 练习

- 1. (*2.5) 取一个C程序并将它转为C++程序; 列出其中使用的所有各类非C++结构, 并确定它们是否合法的ANSI C结构。首先将程序转为严格的ANSI C(加上原型等), 而后转到C++。估计将一个100000行的C程序转为C++所需要的时间。

2. (*2.5) 写一个程序帮助做从C程序到C++的转化，让它重新命名那些已经是C++关键字的变量，用`new`的使用取代对`malloc()`的调用，等等。提示：不要试图完成一项完美的工作。
3. (*2) 将一个C风格的C++程序（可能就是新转过来的C程序）里所有对`malloc()`的调用都换成`new`的使用。提示：B.4[8~9]。
4. (*2.5) 尽可能减少C风格的C++程序（可能就是新转过来的C程序）里所用的宏、全局变量、未初始化的变量和强制。
5. (*3) 取一个从C通过蛮力转换的结果C++程序，从作为C++程序的角度对它加以批判，考虑信息的局部化、抽象、可读性、可扩充性、部分的潜在重用。基于这个批判对程序做各种意义重大的修改。
6. (*2) 取一个小的（例如500行）C++程序，将它转换到C。比较原程序与结果程序的大小，或许再比较它们的可维护性。
7. (*3) 写一小组测试程序去确定一个C++实现是否包括“最后的”标准特征。例如，定义在for语句初始部分的变量的作用域是什么（B.3.6节）？是否支持默认模板参数（B.3.5节）？是否支持成员模板（13.6.2节）？是否支持基于参数的查找（8.2.6节）？提示：B.2.4节。
8. (*2.5) 取一个使用了`<X.h>`头文件的C++程序，将它转为使用`<X>`和`<cX>`头文件的程序。尽可能少用使用指令（`using-directive`）。

附录C 技术细节

在心灵和宇宙最核心的深处，
存在着一个理由。
——Slartibartfast

标准允诺了什么——字符集——整数文字量——常量表达式——提升和转换——多维数组——域和联合——存储管理——废料收集——名字空间——访问控制——到数据成员的指针——模板——*static*成员——*friend*——模板作为模板参数——模板参数推断——*typename*和*template*限定——实例化——名字约束——模板和名字空间——显式实例化——忠告

C.1 引言和概述

本章介绍一些技术细节和相关实例，它们都不能很好地放进我对C++语言特征及其使用的展示之中。当你写程序时，特别是阅读那些用到它们的代码时，这里所介绍的细节也可能是很重要的。当然，我把它们看做是技术细节，因此不应该让它们干扰了学生们学习使用C++的主要工作，也不应让它们干扰了程序员在C++里尽可能清晰而直接地表述思想的主要工作。

C.2 标准

与普遍的信念相反，严格地坚持C++语言和库的标准并不能保证好代码，甚至不能保证可移植的代码。标准没有说一段代码是好代码还是坏代码；它只是说了程序员对一个实现可以依靠或者不可以依靠哪些东西。人们可以写出符合标准的绝对糟糕的程序，而且现实世界中的大部分程序都依赖于一些标准中并未覆盖的特征。

标准中把许多重要事项宣布为由实现定义。这也就意味着每个实现必须为该种结构提供一个特定的、定义良好的行为方式，并必须将它写入文档。例如，

```
unsigned char c1 = 64;           // 有良好定义：char至少有8位，总能保存64
unsigned char c2 = 1256;         // 实现定义的：如果char只有8位就会出现截断
```

*c1*的初始化具有良好的定义，因为一个*char*至少必须有8位。而对*c2*初始化的行为就要由实现定义，因为*char*的确切位数由实现定义。如果*char*只有8位，那么值1256就会被截断到232（C.6.2.1节）。由实现定义的大部分特征都与用于运行程序的硬件的特征有关。

在写实际程序的时候，通常都需要依赖于某些由实现定义的行为。这种行为是我们为了能在大范围的系统之上有效操作所必须付出的代价。例如，如果所有的*char*都是8位，而且所有整数都是32位，这个语言就会简单许多。但是，16位和32位的字符也很常见，整数也可能更大而无法放入32位中。例如，今天的许多计算机都有着超过32G字节的磁盘，表示磁盘地址就需要用48位或者64位的整数。

为了达到最大程度的可移植性，一种明智的做法是让我们所依赖的由实现定义的特征明确化，将更微妙的实例孤立到程序里一些清楚标明的部分之中。一种典型的实际做法就是将所有对硬件的依赖性表述为一些常量和类型定义，放到某个头文件里。正是为支持这类技术，标准库提供了 `numeric_limits` (22.2节)。

无定义的行为则是更严重的。如果标准宣称某种结构是无定义的 (undefined)，那么实现就不必为它提供任何合理的行为。在典型情况下，某些明显的实现技术将导致使用了无定义特征的程序具有极坏的行为方式。例如，

```
const int size = 4*1024;
char page[size];

void f()
{
    page[size+size] = 7; // 无定义
}
```

这一代码段的可能后果包括改写了某些无关的数据或者触发了硬件的错误/异常等。具体实现并不需要去选定某种可能的后果。在使用了强有力的优化系统的地方，无定义行为的实际影响可能变得更加无法预计。如果存在一组可行的而且容易实现的选择方式，这个特征就会被宣布为由实现定义的，而不会是无定义的。

花上一些时间和努力，保证程序里没有使用任何标准所宣称的无定义的东西是非常值得的。对于许多情况，都存在着能提供帮助的工具。

C.3 字符集

本书中的例子都是用国际的7位字符集ISO 646-1983美国变形ASCII (ANSI3.4-1968)写出的。这可能给那些在具有不同字符集的环境里使用C++的人带来三个问题：

- [1] ASCII所包含的一些标点字符和运算符符号 (例如 `]`、`{` 和 `!`) 在某些字符集里将无法使用。
- [2] 我们需要为那些没有方便的字符表示形式的字符 (如换行和“具有值17的字符”) 确定一种记述形式。
- [3] 除英语之外其他语言里包含了一些ASCII所没有的字符，如 ζ 、 \AA 和 Π 等。

C.3.1 受限的字符集

ASCII里的特殊字符 `[`、`]`、`{`、`}`、`|`、`\` 占据了ISO中指定作为字母的字符位置。在大部分欧洲国家的ISO-646字符集中，这些位置由一些英文字母表中没有的字母占据着。例如，丹麦国家字符集用它们表示元音 \AA 、 \ae 、 \O 、 \o 、 \r 和 \a 。没有它们将无法写出任何有价值的丹麦文本。

这里提供了一组三联符，以使各个国家字符能通过一个标准的最小字符集，以一种真正可移植的方式表示。这对于交换程序是很有用的，但却使人们阅读程序非常困难。自然，对C++程序员而言，这个问题的长远解决方案是取得那种既能支持其国家字符集，又能很好地支持C++的设备。可惜的是这种方式对某些人不可行，新设备的引进也可能是一个极其缓慢的过程。为了帮助程序员应付不完全的字符集，C++提供了几种替代的方式：

关键字	二联符	三联符
<i>and</i>	&&	<% { ??= #
<i>and_eq</i>	&=	%> } ??([
<i>bitand</i>	&	<: [??< {
<i>bitor</i>		:>] ??/ \
<i>compl</i>	~	%: # ??)]
<i>not</i>	!	%:%: ## ??> }
<i>or</i>		??' ^
<i>or_eq</i>	=	??!
<i>xor</i>	^	??- ~
<i>xor_eq</i>	^=	
<i>not_eq</i>	!=	

用关键字和二联符写出的程序远比用三联符写出的等价程序容易读得多。然而，如果缺少像 { 这样的字符，还是需要用三联符将“缺少的”字符放入字符串或者字符常量里。例如，'{' 将变成 '??<'。

也有些人喜欢用如 *and* 一类的关键字作为其运算符的习惯记法。

C.3.2 转义字符

若干字符有标准的名字，采用反斜线字符 \ 作为转义字符：

名 字	ASCII名字	C++名字
换行符	NL (LF)	\n
水平制表符	HT	\t
垂直制表符	VT	\v
退格符	BS	\b
回车符	CR	\r
换页符	FF	\f
警铃符	BEL	\a
反斜线符	\	\\
问号	?	\?
单引号	'	\'
双引号	"	\"
八进制数	ooo	\ooo
十六进制数	hhh	\xhhh ...

虽然有这种表现形式，但它们实际上都是单个字符。

可以将一个字符表示为一个、两个或者三个数字的八进制数（\ 后随八进制数字）或者十六进制数（\x 后随十六进制数字）。序列中十六进制数字的个数没有限制。八进制或者十六进制数字序列分别由第一个不是八进制或者十六进制数字的字符结束。例如，

八进制	十六进制	十进制	ASCH
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	'0'
'\137'	'\x05f'	95	'_'

这就使我们可能表示机器字符集中的每个字符，特别是将这些字符嵌入字符串中（5.2.2节）。

采用数值写法写任何字符，都会阻碍程序在采用不同字符集的机器之间移植。

将多于一个字符括在一个字符文字量里也是可能的，例如写 `'ab'`。这是一种陈旧用法，依赖于实现，最好是避免。

在采用八进制写法将数值形式常量嵌入字符串时，最明智的方式是一个数只用三个字符。如不特别小心，这种记法将很难读清楚，无论这种常量之后是不是数字。对于十六进制应该总用两个数字。考虑下面的实例：

```
char v1[] = "\xah\129";    // 6个字符, 'a'\xa'h'\12'\9'\0'
char v2[] = "\xah\127";    // 5个字符, 'a'\xa'h'\127'\0'
char v3[] = "\xad\127";    // 4个字符, 'a'\xad'\127'\0'
char v4[] = "\xad0127";    // 5个字符, 'a'\xad'\012'\7'\0'
```

C.3.3 大字符集

C++程序可以用某种远远比127个字符的ASCII集合更大的字符集写出，并以该种形式提供给用户。在支持大字符集的实现里，标识符、注释、字符常量、字符串都可以包含诸如 α 、 β 、 Γ 一类的字符。但是，为了可移植性，这个实现必定要将这些字符映射到每个C++用户都可以使用的字符的某种编码中。原则上说，到C++基本源程序字符集（也就是本书所用的集合）的这个翻译应该在编译器做其他任何处理之前完成，所以它不会影响程序的语义。

C++直接支持从大字符集到小字符集的标准编码，所采用方式是4个或者8个十六进制数字的序列：

```
universal-character-name:
    \U XXXXXXXXXX
    \u XXXX
```

这里的X表示一个十六进制数字，例如 `\ule2b`。短形式 `\uXXXX` 等价于 `\U0000XXXX`。十六进制数字的个数不是4个或者8个时将作为词法错误。

程序员可以直接采用这种字符编码。然而，这些形式主要是作为实现的一种内部手段，以便能用小的字符集来处理程序员所见的大字符集里的字符。

如果你对于标识符使用了依赖于特定环境提供的扩充字符集，程序的可移植性将会下降。除非人们能理解用做标识符和注释的自然语言，否则这个程序就会很难读。因此，为全世界使用的程序最好还是坚持用英语和ASCII。

C.3.4 有符号和无符号的字符

普通`char`究竟有符号还是无符号，此事由实现定义。这也导致可能出现讨厌的、令人诧异的和依赖于实现的情况。例如，

```
char c = 255; // 255是“全1”，十六进制的0xFF
int i = c;
```

`i`的值是什么？不幸的是这里没有明确的回答。在我所知道的所有实现上，答案都依赖于将“全1”`char`的二进制模式扩展到`int`的意义。在SGI Challenge机器上`char`没有符号，所以答案是255。在Sun SPARC或者IBM PC上的`char`有符号，所以答案是-1。在这种情况下，编译器可能对于将文字量255转换到-1提出警告，但是C++无法提供某种一般性的机制来检查这类问题。一种解决问题的办法是避免普通的`char`，只用特定的`char`类型。可惜的是有些标准库函

数只接受普通`char`类型，例如`strcmp()`（20.4.1节）。

`char`的行为必然或者等同于`signed char`，或者等同于`unsigned char`。然而，因为这三个`char`类型不同，所以你不能混用这三种类型的指针。例如，

```
void f(char c, signed char sc, unsigned char uc)
{
    char* pc = &uc;           // 错误：没有指针转换
    signed char* psc = pc;      // 错误：没有指针转换
    unsigned char* puc = pc;    // 错误：没有指针转换
    psc = puc;                  // 错误：没有指针转换
}
```

这三种`char`类型的变量可以自由地相互赋值。当然，如果给有符号的`char`赋一个过大的值（C.6.2.1节），结果仍然是无定义的。例如，

```
void f(char c, signed char sc, unsigned char uc)
{
    c = 255; // 如果普通char有符号且为8位，则由实现定义
    c = sc;  // 可以
    c = uc;  // 如果普通char有符号且uc值过大，则由实现定义
    sc = uc; // 如果uc的值过大，则由实现定义
    uc = sc; // 可以：转换到unsigned值
    sc = c;  // 如果普通char无符号且c值过大，则由实现定义
    uc = c;  // 可以：转换到unsigned值
}
```

如果你始终采用普通的`char`，这些潜在问题都不会出现。

C.4 整数文字量的类型

一般说，整数文字量的类型依赖于它的形式、值和后缀：

- 如果它是十进制形式且无后缀，在`int`、`long int`、`unsigned long int`中的哪个是第一个可以表示这个文字量的值的类型，该文字量就属于那个类型。
- 如果是八进制或十六进制形式且无后缀，在`int`、`unsigned int`、`long int`、`unsigned long int`中哪个是第一个可以表示这个文字量的值的类型，该文字量就属于那个类型。
- 如果它有后缀`u`或者`U`，在`unsigned int`、`unsigned long int`中哪个是第一个可以表示这个文字量的值的类型，该文字量就属于那个类型。
- 如果它有后缀`l`或者`L`，在`long int`、`unsigned long int`中哪个是第一个可以表示这个文字量的值的类型，该文字量就属于那个类型。
- 如果它的后缀是`ul`、`lu`、`uL`、`Lu`、`Ul`、`lU`、`UL`或`LU`，则它的类型为`unsigned long int`。

例如，`100000`在32位`int`的机器上类型为`int`，而在16位`int`和32位`long`的机器上类型就是`long int`。类似的，`0XA000`在32位`int`的机器上类型为`int`，而在16位`int`的机器上类型为`unsigned int`。可以通过使用后缀来避免这种实现依赖性：`100000L`在所有机器上的类型都是`long int`，`0XA000U`在所有机器上的类型都是`unsigned int`。

C.5 常量表达式

在一些位置，例如数组界（5.2节），`case`标号（6.3.2节），枚举符的初始式（4.8节）等，C++要求写常量表达式（`constant expression`）。常量表达式求出一个整数的或者一个枚举的常

量。这种表达式由文字量（4.3.1节、4.4.1节、4.5.1节）、枚举符（4.8节）和用常量表达式初始化的`const`组成。在模板中也可以使用一个整数的模板参数（C.13.1节）。浮点文字量（4.5.1节）也可以用，但需要显式地转换到某个整数类型。函数、类对象、指针和引用只能作为`sizeof`运算符（6.2节）的运算对象使用。

直观地看，常量表达式是一些简单表达式，它们能在程序连接（9.1节）和运行之前由编译器求值。

C.6 隐式类型转换

在赋值和表达式里，整类型和浮点类型（4.1.1节）可以随意混合使用。只要可能，就进行不丢失信息的值转换。不幸的是，也会隐式地执行一些破坏值的转换。本节将介绍转换规则、转换中的问题以及解决它们的方法。

C.6.1 提升

维持值不变的隐式转换也常常被称为提升。在执行一个算术运算之前，要通过整数提升从短的整数类型创建出`int`值。注意，这种提升将不会得到`long`（除非运算对象是`wchar_t`或者某个枚举符，其值本身就大于`int`）。这也反映了在C里面这种提升的原意：将运算对象转到做对于算术运算“最自然的”类型。

整数提升将：

- 如果`int`能够表示原类型的所有值的话，就把`char`、`signed char`、`unsigned char`、`short int`或`unsigned short int`转换为一个`int`；否则就转换为一个`unsigned int`。
- 把`wchar_t`（4.3节）或者枚举类型（4.8节）转换到下述类型中第一个能表示`wchar_t`或枚举类型中所有值的类型：`int`、`unsigned int`、`long`或者`unsigned long`。
- 如果`int`可以表示一个位域的所有值，就将这个位域（C.8.1节）转换到`int`。否则，如果`unsigned int`能够表示它的所有值，就转换到`unsigned int`。否则就不对它做任何提升。
- 把`bool`转换到`int`，`false`转为0，`true`转为1。

提升被用做常规算术转换的一部分（C.6.3节）。

C.6.2 转换

基础类型互相转换的多种方式使人眼花缭乱。按照我的看法，允许的转换方式实在太多了。例如，

```
void f(double d)
{
    char c = d;    // 小心：从双精度浮点数到char的转换
}
```

在写代码时，你应始终坚持的目标是避免无定义行为和不声不响地抛弃信息的转换。编译器可以对许多有疑问的转换提出警告。幸运的是许多编译器确实这样做了。

C.6.2.1 整数转换

一个整数可以转换到另一个整数类型。一个枚举符可以转换到一个整数类型。

如果目标类型是`unsigned`，结果值就是从源值中取出正好等于目标类型所需要的那么多位（必要时将多余的高位丢掉）。说得更精确些，结果就是源值的以2的 n 次方为模的那个最小

的无符号同余值，这里的 n 是表示目标值所用的二进制位数。例如，

```
unsigned char uc = 1023; // 二进制111111111; uc变成11111111, 即是255
```

如果目标类型是`signed`，如果源值在结果类型中可以表示，那么值就不变；否则，结果值由实现定义：

```
signed char sc = 1023; // 实现定义
```

可能值是127或者-1（C.3.4节）。

布尔值和枚举值可以隐式地转换到它们的整数等价值（4.2节、4.8节）。

C.6.2.2 浮点转换

一个浮点值可以转换到另一个浮点类型。如果源值在结果类型中有准确的表示，结果还是原来的数值。如果源值位于结果类型的两个相邻值之间，结果就是这两个值之一。否则，转换的行为无定义。例如，

```
float f = FLT_MAX; // 最大的float值
double d = f; // 可以: d == f
float f2 = d; // 可以: f2 == f
double d3 = DBL_MAX; // 最大的double值
float f3 = d3; // 如果FLT_MAX < DBL_MAX则无定义
```

C.6.2.3 指针与引用转换

任何指向某个对象的指针可以隐式地转换到`void*`（5.6节）。一个指向派生类的指针（或引用）可以隐式地转换为指向它的某个可以访问的无歧义性的基类的指针（或引用；12.2节）。注意，指向函数的指针或者指向成员的指针不能隐式地转换到`void*`。

一个能求出值0的常量表达式（C.5节）可以隐式地转换到任意指针或者指向成员的指针（5.1.1节）。例如，

```
int* p =
    ! ! ! ! !
    ! ! ! ! !
    ! ! ! ! !
    ! ! ! ! !;
```

一个`T*`可以隐式地转换到`const T*`（5.4.1节）。类似的，一个`T&`可以隐式地转换到一个`const T&`。

C.6.2.4 到成员的指针的转换

到成员的指针和引用可以按照15.5.1节所描述的方式转换。

C.6.2.5 布尔转换

指针、整数和浮点值可以隐式地转换为`bool`（4.2节）。非0值转换为`true`，0值转换为`false`。例如，

```
void f(int* p, int i)
{
    bool is_not_zero = p; // 如果 p != 0 则为 true
    bool b2 = i; // 如果 i != 0 则为 true
}
```

C.6.2.6 浮点与整数间转换

当某个浮点值转换到整数值时，小数部分将丢掉。换句话说，从浮点类型到整数类型的转换就是截断。例如，`int(1.6)`的值是1。如果转换结果无法在目标类型里表示，截断的行为

无定义。例如，

```
int i = 2.7;           // i变成2
char b = 2000.7;       // 对8位char无定义：2000不能在8位char里表示
```

在硬件允许的情况下，从整数到浮点类型的转换都是数学上正确的。如果在浮点数类型中无法表示有关整数的精确值，那么就会出现精度丢失的现象。例如，

```
int i = float(1234567890);
```

在某个int和float都用32位表示的机器上，i的值为1234567936。

很清楚，最好是避免所有破坏值的隐式转换。事实上，编译器能够检查并警告某些明显的很危险的转换，例如从float到int或者long int到char。然而，一般性的编译时检查是不实际的，所以程序员必须小心。在仅仅“小心”还不够的那些地方，程序员就应该插入显式的检查。例如，

```
class check_failed { };

char checked(int i)
{
    char c = i;                // 警告：不可移植 (C.6.2.1节)
    if (i != c) throw check_failed();
    return c;
}

void my_code(int i)
{
    char c = checked(i);
    // ...
}
```

要想以某种保证可移植的方式完成截断，需要使用numeric_limits (22.2节)。

C.6.3 普通算术转换

对二元运算符的两个运算对象将执行下述转换，使它们都变到一个公共类型，该类型也被作为运算的结果类型：

- [1] 如果某个运算对象的类型为long double，则另一个也转换为long double。
- 否则，如果某个运算对象是double，则另一个也转换为double。
- 否则，如果某个运算对象是float，则另一个也转换为float。
- 否则对两个运算对象都做整数提升 (C.6.1节)。
- [2] 而后，如果某个运算对象的类型为unsigned long，则另一个也转换为unsigned long。
- 否则，如果某个运算对象是long int而另一个是unsigned int，那么如果long int能够表示unsigned int的所有值，则那个unsigned int转换为long int；否则就将两者都转换为unsigned long int。
- 否则，如果某个运算对象是long，则另一个也转换为long。
- 否则，如果某个运算对象是unsigned，则另一个也转换为unsigned。
- 否则，两个运算对象都是int。

C.7 多维数组

需要向量的向量、向量的向量的向量等也是很常见的。问题是如何在C++里表示这些多维

向量。这里我将首先展示如何使用标准库的`vector`类，而后再介绍多维数组，这类数组也出现在只使用内部功能的C和C++程序里。

C.7.1 向量

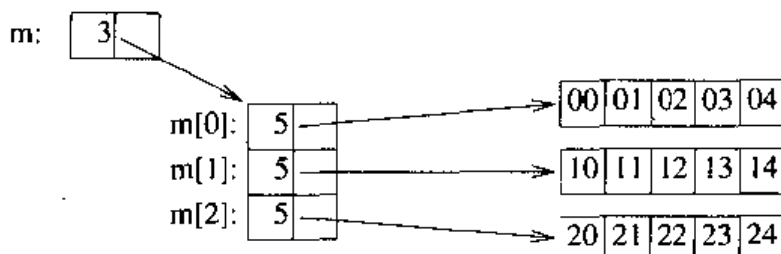
标准`vector`（16.3节）提供了一种非常一般的解决方案：

```
vector< vector<int> > m(3, vector<int>(5));
```

这样就创建了一个包含三个向量的向量，其中的每个向量里包含5个整数。这15个元素中都保存着默认值0。我们可以按如下方式给这些整数元素赋新值：

```
void init_m()
{
    for (int i = 0; i < m.size(); i++)
        for (int j = 0; j < m[i].size(); j++) m[i][j] = 10*i+j;
}
```

用图形表示：



每个向量实现为一个到它的元素的指针加上一个元素个数。元素通常保存在一个数组里。为说明起见，我让每个`int`值表示它本身的坐标位置。

访问元素的方式是通过两次下标。例如，`m[i][j]`是第`i`个向量的第`j`个元素。我们可以如下打印`m`：

```
void print_m()
{
    for (int i = 0; i < m.size(); i++) {
        for (int j = 0; j < m[i].size(); j++) cout << m[i][j] << ' ';
        cout << '\n';
    }
}
```

这将给出

```

0   1   2   3   4
10  11  12  13  14
20  21  22  23  24
  
```

注意，`m`是一个向量的向量而不是一个多维向量。特别地，可以重新确定其元素的大小（16.3.4节）。例如，

```
void reshape_m(int ns)
{
    for (int i = 0; i < m.size(); i++) m[i].resize(ns);
}
```

在`vector< vector<int> >`里的各个`vector<int>`的大小也不必相同。

C.7.2 数组

内部数组是出错的一个主要根源，特别是用它们创建多维数组时。对于初学者，它们也

是造成混乱的一个主要根源。只要可能,就应该用`vector`、`list`、`valarray`、`string`等。

多维数组用数组的数组表示。下面定义了一个3乘5的数组:

```
int ma[3][5]; // 3个数组, 每个包含5个int
```

我们可以如下方式对`ma`初始化:

```
void init_ma()
{
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 5; j++) ma[i][j] = 10*i+j;
}
```

或图示如下:

```
ma:  

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 01 | 02 | 03 | 04 | 10 | 11 | 12 | 13 | 14 | 20 | 21 | 22 | 23 | 24 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|


```

`ma`就是15个`int`的数组,我们访问它的方式就像它是3个数组,每个包含5个元素。特别地,在存储器中并不存在任何代表着矩阵`ma`的对象——只存储了所有的元素。维数3和5只存在于被编译的源文件里。在写代码时,记住这些并在需要时提供正确的维情况是我们的工作。例如,我们可能以如下方式打印出`ma`:

```
void print_ma()
{
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) cout << ma[i][j] << ' ';
        cout << '\n';
    }
}
```

在某些语言里使用的逗号记法不能用于C++,因为逗号(,)是这里的序列运算符(6.2.2节)。幸运的是,大部分错误都可以被编译器检查出来。例如,

```
int bad[3,5];           // 错误: 常量表达式里不允许逗号
int good[3][5];         // 3个数组, 每个包含5个元素
int ouch = good[1,4];    // 错误: 用int* 初始化int (good[1,4] 的意思是good[4], 它是一个int*)
int nice = good[1][4];
```

C.7.3 传递多维数组

现在考虑如何定义对二维矩阵进行操作的函数。如果在编译时已知两个维的情况,那么就不存在任何问题:

```
void print_m35(int m[3][5])
{
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 5; j++) cout << m[i][j] << ' ';
        cout << '\n';
    }
}
```

用多维数组表示的矩阵将作为指针传递(而不是复制;5.3节)。数组的第一个维与确定元素位置的问题无关,它不过是陈述了有多少个(这里是3个)适当类型(这里是`int[5]`)的元素。举例来说,查看前面`ma`的表示方式,可以注意到,只要我们知道第二个维是5,我们就可以对任意的`i`确定`ma[i][5]`的位置。因此可以将第一个维作为参数传递:

```
void print_mi5(int m[][5], int dim1)
{
    for (int i = 0; i < dim1; i++) {
        for (int j = 0; j < 5; j++) cout << m[i][j] << ' ';
        cout << '\n';
    }
}
```

最困难的情况是需要传递两个维的值。“明显的”解决方案根本无法工作：

```
void print_mij(int m[][ ], int dim1, int dim2) // 不像许多人想像的那样工作
{
    for (int i = 0; i < dim1; i++) {
        for (int j = 0; j < dim2; j++) cout << m[i][j] << ' ';    // 出乎意料!
        cout << '\n';
    }
}
```

首先，参数声明 `m[][]` 本身就非法。因为，为了确定元素的位置，多维数组的第二个以后的维都必须知道。其次，表达式 `m[i][j]` 将被（正确地）解释为 `*(*(m + i) + j)`，虽然这不大像程序员所期望的。一种正确的解决方法是：

```
void print_mij(int* m, int dim1, int dim2)
{
    for (int i = 0; i < dim1; i++) {
        for (int j = 0; j < dim2; j++) cout << m[i*dim2+j] << ' '; // 不易理解
        cout << '\n';
    }
}
```

在 `print_mij()` 里访问数组成员所用的表达式，正好等价于在知道了维的情况时，编译器所产生的表达式。

要调用这个函数，我们需要用普通指针传递矩阵：

```
int main()
{
    int v[3][5] = { {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24} };
    print_m35(v);
    print_mi5(v, 3);
    print_mij(&v[0][0], 3, 5);
}
```

请注意最后一个调用中所用的 `&v[0][0]`；`v[0]` 与它等价，但写 `v` 将是个类型错误。最好还是避免这类微妙而难搞的代码。如果你必须直接处理多维数组，请考虑将所有依赖于它的代码封装起来。按照这种方式，你就可以比较容易地作为下一个程序员接触这些代码了。提供一种多维数组类型，带着某种完好的下标运算，将能使大多数用户不必去关心数据在数组中的布局问题（22.4.6节）。

标准 `vector`（16.3节）不会受到这类问题的滋扰。

C.8 节约空间

在做复杂应用的程序设计时，常常会有某个时候，你所需要的空间比可用的或者可以负担的空间更大。存在两种从可用空间中挤压出更多位置的方式：

[1] 把多于一个小对象存入一个字节里。

[2] 用同一位置在不同时刻保存不同的对象。

前者可以通过位域实现，后者可以通过联合。下面几小节将介绍这些结构。位域和联合的许多应用都完全是为了优化，而这些优化往往都基于某些不可移植的关于存储布局的假设。因此，程序员在使用它们之前应该三思。更好的方式常常是改变管理数据的方式，例如，更多地依赖于动态分配的存储（6.2.6节），更少依靠预分配的（静态）存储等。

C.8.1 位域

用整个的一个字节（*char*或*bool*）去表示一个二进制变量（例如一个on/off开关）看起来是太铺张浪费，而*char*已经是在C++里可以独立分配与寻址的最小对象了（5.1节）。可能将几个这种小变量绑在一起，作为*struct*的域，将成员定义为域的方式就是描述它所占据的二进制位的个数。也允许无名的域，它们不影响有名域的意义，但它们可用于以一种不依赖于机器的方式改善*struct*的布局：

```
struct PPN {           // R6000物理页编号（Physical Page Number）
    unsigned int PFN : 22; // 页帧编号（Page Frame Number）
    int : 3;              // 不用
    unsigned int CCA : 3;  // 高速缓存一致性算法
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

这个例子也显示了域的另一种主要用途：为某种由外部确定的布局中的各个部分命名。域必须是整型或者枚举类型（4.1.1节）。不能取得域的地址，除此之外，域就完全可以像其他变量一样使用了。注意，一个*bool*域真的可以用一个二进制位表示。在某个操作系统的核心或者排错系统里，类型*PPN*可能以如下方式使用：

```
void part_of_VM_system (PPN* p)
{
    // ...

    if (p->dirty) { // 内容变了
        // 复制到磁盘
        p->dirty = 0;
    }

    // ...
}
```

令人感到意外的是，采用域将几个变量装进一个字节未必就能节省空间。这样做确实能减少数据空间，但是在大多数机器上，用于操作这些变量的代码的规模将增大。人们已知有一些程序，在将二进制变量从位域表示改为字符表示后，程序规模显著地减小了。进一步说，访问一个*char*或者*int*在速度上通常也比访问位域快许多。位域也就是一种使用按位逻辑运算符（6.2.4节）的方便形式，可用于从机器字中提取部分信息或者将信息放入机器字的一部分里。

C.8.2 联合

一个*union*（联合）是一个*struct*，但它的所有成员被分配在同一个地址，因此这个*union*

只占据它最大的成员所需要的那么多空间。自然，一个 **union** 在每个时刻只能保存它的一个成员的值。举个例子，考虑一个符号表项，表项中保存一个名字和一个值：

```
enum Type { S, I };

struct Entry {
    char* name;
    Type t;
    char* s; // 如果 t == S 则用 s
    int i;    // 如果 t == I 则用 i
};

void f(Entry* p)
{
    if (p->t == S) cout << p->s;
    // ...
}
```

成员 *s* 和 *i* 绝不会同时使用，所以这里浪费了空间。将两者描述成 **union** 的成员就可以很容易地将空间找回来，像下面这样做：

```
union Value {
    char* s;
    int i;
};
```

语言并不跟踪有关一个 **union** 里到底存着哪种值的信息，程序员需要采用如下方式工作：

```
struct Entry {
    char* name;
    Type t;
    Value v; // 如果 t == S 则用 v.s，如果 t == I 则用 v.i
};

void f(Entry* p)
{
    if (p->t == S) cout << p->v.s;
    // ...
}
```

遗憾的是，引进 **union** 就迫使我们去重写代码，必须写 *v.s* 而不是 *s*。采用匿名联合可以避免这个问题。匿名联合也是一个联合，但它没有名字，因此也不定义类型。这种联合的作用只是保证其成员可以分配在同样的地址中：

```
struct Entry {
    char* name;
    Type t;
    union {
        char* s; // 如果 t == S 则用 s
        int i;   // 如果 t == I 则用 i
    };
};

void f(Entry* p)
{
    if (p->t == S) cout << p->s;
    // ...
}
```

这就使使用 **Entry** 的代码可以保持不变了。

正确地使用`union`，保证读它的值时总是经由写值时所用的成员，也就是一种纯粹的优化。然而，要保证总能按照这种方式使用`union`可不那么容易，而且误用将引进微妙的错误。为避免错误，可以将`union`封装起来，以便保证类型域与对`union`成员的访问之间的相互对应（10.6[20]）。

联合有时也被错误地用于“类型转换”。这种错误使用主要是由那些在没有显式类型转换机制的语言里训练出来的程序员做的，在那里必须采用欺骗手段。例如，下面是将`int`“转换”到`int*`，简单地假定它们按位等价：

```
union Fudge {
    int i;
    int* p;
};

int* cheat(int i)
{
    Fudge a;
    a.i = i;
    return a.p;    // 很坏的法
```

这根本就不是一个转换。在某些机器上，`int`和`int*`所占存储空间的数量并不一样，而在另一些机器上，整数不能有奇数地址。`union`的这种使用是极其危险且不可移植的，而且存在着显式的可移植的描述类型转换的方式（6.2.7节）。

偶尔人们也有意使用联合去避免类型转换。举个例子，某人可以借助于`Fudge`去查找指针0的表示：

```
int main()
{
    Fudge foo;
    foo.p = 0;
    cout << "the integer value of the pointer 0 is " << foo.i << '\n';
}
```

C.8.3 联合与类

在许多不那么简单的`union`里，常常会有某些成员比其他最常用的成员大得多。由于`union`的大小至少与它最大的成员一样，这样就会浪费许多空间。采用一组派生类（而不用`union`）常常可以避免这种浪费。

一个有构造函数、析构函数或者复制操作的类不能作为`union`成员的类型（10.4.12节），因为编译器无法确定应该销毁什么成员。

C.9 存储管理

在C++里使用存储有如下几种基本方式：

静态存储。连接器在这里为程序的运行分配对象。全局的和名字空间的变量、`static`类变量（10.2.4节）和函数里的`static`变量都在静态存储分配。在静态存储分配的变量只构造一次，它们一直持续存在到程序结束，总具有同一个地址。静态变量在使用线程的（共享地址空间的并行）程序里有可能成为问题，因为它们是共享的，为了正确访问就需要锁定。

自动存储。在这里分配函数参数和局部变量。函数或者块的每个项都有自己的副本。这

种对象将自动创建和销毁，这也就是自由存储的名字的由来。自动存储也被说成是“放在堆栈里”。如果你真的希望使这种情况更明显，C++提供了一个冗余的关键字`auto`。

自由存储。程序在这里显式地要求创建对象，在用过它们之后又可以重新释放其存储（通过`new`和`delete`）。当程序需要更多的存储时，`new`将向操作系统发出请求。典型情况下，在一个程序的整个运行期间，自由存储（也称为动态存储或者堆）将不断扩大，因为它不会把存储返还操作系统供其他程序使用。

仅就程序员的关注范围而言，自动存储和静态存储是以简单、明确和隐式的方式使用的，更有意思的问题是如何管理自由存储。分配操作比较简单（通过`new`），但是，除非我们有某种统一的策略将存储交还自由存储管理程序，否则存储总会被充满——特别是对那些长时间运行的程序。

最简单的策略就是利用自动对象去管理自由存储中对应的对象。因此，许多容器都被实现为到自由存储对象的句柄（25.7节）。例如，一个自动的`String`（11.12节）管理着自由存储里的一个字符序列，当自己退出作用域时就会自动释放那些存储。所有标准容器（16.3节，第17章，第20章，22.4节）都能按这种方式方便地实现。

C.9.1 自动废料收集

如果这种正规的方式不充分，程序员还可以使用一种能够找出无引用对象并回收它们的存储，以便存放新对象的存储管理器。这一过程通常被称为自动废料收集，或简称废料收集。很自然，这种存储管理器也被称为废料收集器。

废料收集的基本思想就是，程序里的任何不能再引用的对象也就不能再访问了，因此，它的存储就可以安全地用于存放新的对象了。例如，

```
void f()
{
    int* p = new int;
    p = 0;
    char* q = new char;
}
```

在这里，赋值`p = 0`使那个`int`无法再引用，因此，它的存储就可以用于别的新对象；所以随后的`char`就可以分配在与上述`int`同样的存储里，使`q`保存着与`p`开始时同样的值。

标准本身并不要求一个实现提供废料收集器，但废料收集器在C++里的使用正在增长，特别是在那些使用它们的代价比手工管理自由存储区更为合算的领域中。在比较代价时，需要考虑运行时间、存储的使用、可靠性、可移植性、程序设计的资金成本、废料收集器的资金成本以及性能的可预见性等。

C.9.1.1 伪装的指针

说一个对象无法引用是什么意思呢？考虑

```
void f()
{
    int* p = new int;
    long i1 = reinterpret_cast<long>(p) & 0xFFFF0000;
    long i2 = reinterpret_cast<long>(p) & 0x0000FFFF;
    p = 0;
    // 程序点#1：这里不再有指向那个int的指针了
```

```

    p = reinterpret_cast<int*>(i1 | i2);
    // 现在又有到那个int的指针了
}

```

在程序中作为非指针存储的指针常常被称做“伪装的指针”。特别是，上面原本存储在`p`里的指针被伪装后存入整型`i1`和`i2`里。当然，废料收集器并不需要关注伪装的指针。如果废料收集器在位置 `#1` 运行，它就可以将保存那个`int`的空间收回。事实上，即使没有废料收集器，这个程序也不能保证工作，因为用`reinterpret_cast`在整数与指针间进行转换的意义，在最好情况下也是由实现定义的。

能保存指针值和非指针值的`union`给废料收集器提出了一个特殊的问题。一般说，无法断定这样的—个`union`里是否保存着一个指针。考虑

```

union U {           // 有指针和非指针成员的union
    int* p;
    int i;
};

void f(U u, U u2, U u3)
{
    u.p = new int;
    u2.i = 999999;
    u.i = 8;
    // ...
}

```

比较安全的假设是认为出现在这种`union`里的任何值都是指针值。聪明的废料收集器可能工作得更好一些。例如，它可能（对于某个确定的实现）注意到`int`不会分配在奇数地址，没有对象会分配在像`8`这样低的地址中。注意到这些，废料收集器就不会假定包含位置`999999`和`8`的对象正在由`f()`使用了。

C.9.1.2 删除

如果某个实现能够自动收集废料，那么就不再需要用`delete`和`delete[]`运算符去为潜在的重新使用而释放存储了。这样，依靠废料收集器的用户就可以免除使用这些操作的麻烦。然而，除了释放存储之外，`delete`和`delete[]`还要调用析构函数。

在有了废料收集器的情况下，

```
delete p;
```

只是为由`p`所指的—对象调用析构函数（如果有的话），而存储的重新使用则可以推迟到它被收集之时。一次收集大量的对象能帮助限制碎片问题（C.9.1.4节）。对于那些析构函数只是简单完成清除存储的重要情况，这样做也能使删除对象两次变成无害的事情（原本这应是一个严重错误）。

与往常—样，在删除之后访问对象仍然是无定义的。

C.9.1.3 析构函数

当废料收集器准备回收一个对象时，存在着两种可能的选择：

[1] 为该对象调用析构函数（如果有的话）。

[2] 将对象看做是原始的存储（不调用它的析构函数）。

按默认方式，废料收集器应该取选择 [2]，因为由`new`创建的对象如果不`delete`，那么就将永不销毁。这样，我们可以将废料收集器看做—种模拟无限存储区的机制。

也可能设计出一种废料收集器，让它去调用通过某种特殊方式在收集器中“注册”的对象的析构函数。当然，并不存在任何标准的对象“注册”的方式。注意，保证对象的销毁顺序非常重要，也就是说，要保证一个对象的析构函数不会引用另一个已经销毁的对象。没有程序员的帮助，废料收集器很难保证这种顺序。

C.9.1.4 存储碎片

如果被分配和释放的许多对象具有不同的大小，存储区就会碎片化，也就是说，许多存储被很小的无法有效使用的碎片所消耗。产生这种情况的原因是，通用分配器未必总能为一个对象找到一片大小正好合适的存储，使用稍微大一点的存储片就意味着留下更小的存储片。在用简单的分配器运行程序一段时间之后，发现可用存储中的一半被很小的无法再用的片段占据也是很常见的情况。

存在着一些对付碎片问题的技术。最简单的就是为分配器申请一大块存储，并让同样大小的对象使用这种大的存储块（15.3节、19.4.2节）。因为大部分存储分配和释放都是针对很小的对象类型，例如树结点、链接等，这一技术可能非常奏效。分配器也可能有时自动采用类似的技术。在各种情况下，如果大的“块”都具有同样大小（譬如说，一个页面的大小）以使它们本身又可以无碎片地分配和重新分配，碎片就会进一步减少。

存在两种主要风格的废料收集器：

[1] 复制式收集器在存储区里搬动对象，以便将片段空间紧缩到一起。

[2] 保守式收集器分配对象时尽可能减少碎片问题。

从C++的观点看，保守式废料收集器是更合用的，因为要移动对象同时又将指针修改正确将非常困难（对于实用程序或许就不可能）。保守式收集器还能允许C++代码片段与用C等语言写出的代码共存。按照传统，复制式收集器被使用某些语言（如Lisp或Smalltalk）的人们所喜爱，在那些语言里，对象都是通过惟一的指针或者引用间接处理的。无论如何，对于大程序而言，新型的保守式收集器至少与复制式收集器一样有效。在这里，复制的量以及收集器与分页系统之间的交互代价已经变得非常重要了。对于小程序，不使用收集器常常就很理想了，特别是在C++中，因为其中的许多对象都是自动的。

C.10 名字空间

本节介绍名字空间的一些次要问题，它们看起来属于技术细节，然而在讨论中或者实际代码中也频繁出现。

C.10.1 方便与安全

使用声明把一个名字放入局部的作用域，使用指令则不是这样，它简单地使一些名字在它们被声明的作用域里成为可访问的。例如，

```
namespace X {
    int i, j, k;
}

int k;

void f1()
{
    int i = 0;
```



```

    using namespace X; // 使来自X的名字可访问
    i++;               // 局部的i
    j++;               // X::j
    k++;               // 错误: X::k还是全局的k?
    ::k++;             // 全局的k
    X::k++;            // X的k
}

void f2()
{
    int i = 0;
    using X::i;        // 错误: i在f2()里定义两次
    using X::j;
    using X::k;        // 屏蔽全局的k

    i++;
    j++;               // X::j
    k++;               // X::k
}

```

局部声明的名字（无论是通过常规声明还是通过使用声明）将屏蔽非局部的同样名字的声明，而对任何名字的非法重载都会在声明处被检查出来。

请注意*f1()*里*k++*的歧义错误。与被做成在全局作用域里可访问的名字空间里的名字相比，全局的名字并没有得到更高的优先权。这是为防止名字冲突而提供了一种重要保护机制，而且更重要的是保证了通过污染全局名字空间不会取得任何利益。

在通过使用指令将声明大量名字的库变得可以访问时，保证其中在未使用的名字上出现的冲突不是错误，这也是一件很重要的事情。

全局作用域也就是另一个名字空间。全局名字空间仅有的奇特之处就是你在使用显式限时不必提出它的名字。也就是说，*::k*的意思是“到全局名字空间和在全局名字空间中的使用指令里提到的名字空间里去找*k*”。而*X::k*则表示“在名字空间*X*里或者在*X*中的使用指令里提到的名字空间里的*k*”（8.2.8节）。

我希望能看到，与传统的C和C++程序相比，在采用了名字空间的新程序里，全局名字的使用能够大大减少。有关名字空间的规则经过了特别的琢磨，以使与那些细心地不污染全局作用域的人相比，采用全局名字的“懒惰”用户不能占到任何便宜。

C.10.2 名字空间的嵌套

名字空间的最明显使用方式就是把一组声明和定义包装到一个独立的名字空间里：

```

namespace X {
    // 我的所有声明
}

```

一般说，在这些声明中也可以包含名字空间。这样就允许了嵌套名字空间。允许这样做确有实际理由。除此之外，为了简单起见，各种结构都应该能够嵌套，除非有充分的理由说它不应该如此。看下而例子：

```

void h();

namespace X {
    void g();
    // ...
    namespace Y {

```

```

        void f();
        void ff();
        // ...
    }
}

```

普通的作用域和限定规则在这里也都适用：

```

void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();          // 错误：X里无f()
    Y::f();       // ok
}

void h()
{
    f();          // 错误：无全局的f()
    Y::f();       // 错误：无全局的f()
    X::f();       // 错误：X里无f()
    X::Y::f();    // ok
}

```

C.10.3 名字空间与类

一个名字空间就是一个命名的作用域。一个类就是由命名的作用域定义的一个类型，它描述了该类型的对象应该如何创建和使用。由此可见，名字空间是一个比类更简单的概念，在理想情况下，一个类可以定义为一个带有若干额外功能的名字空间。情况几乎就是这样。但名字空间是开放的（8.2.9.3节），而类是封闭的。这种差异源自对类的观察：类需要定义对象的布局，这件事最好是在一个地方完成。还有，使用声明和使用指令只能以极其受限的方式对类使用（15.2.2节）。

当你需要做的事也就是封装起一些名字时，更应该用名字空间而不是类。在这种情况下，根本不需要为类配置的类型检查、创建对象等的功能，简单的名字空间就足够了。

C.11 访问控制

本节将介绍有关访问控制的一些技术示例，作为对15.3节中讨论的补充。

C.11.1 访问成员

考虑

```

class X {
// 默认为private:
    int priv;
protected:
    int prot;
public:
    int publ;
    void m();
};

```

成员 `X::m()` 具有不受限制的访问权:

```
void X::m()
{
    priv = 1; // ok
    prot = 2; // ok
    publ = 3; // ok
}
```

派生类的成员可以访问这里公用的和保护了的成员 (15.3节):

```
class Y : public X {
    void mderived();
};
void Y::mderived()
{
    priv = 1; // 错误: priv为私有
    prot = 2; // 可以: prot是保护的, mderived()是派生类Y的成员
    publ = 3; // 可以: publ为公用
}
```

全局函数只能访问公用成员:

```
void f(Y* p)
{
    p->priv = 1; // 错误: priv为私有
    p->prot = 2; // 错误: prot是保护的, f()不是友元也不是类X或Y的成员
    p->publ = 3; // 可以: publ为公用
}
```

C.11.2 访问基类

与成员一样, 基类也可以声明为 `private`、`protected` 或者 `public`。考虑

```
class X {
public:
    int a;
    // ...
};

class Y1 : public X { };
class Y2 : protected X { };
class Y3 : private X { };
```

因为 `X` 是 `Y1` 的公用基类, 在所有需要之处, 任何函数都可以从 `Y1*` (隐式地) 转换到 `X*`, 以便能访问类 `X` 的公用成员。例如,

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1; // 可以: X是Y1的公用基类
    py1->a = 7; // 可以

    px = py2; // 错误: X是Y2的保护基类
    py2->a = 7; // 错误

    px = py3; // 错误: X是Y3的私有基类
    py3->a = 7; // 错误
}
```

考虑

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(Y1*, Y2*, Y3*); };
```

因为 X 是 $Y2$ 的保护基类，只有 $Y2$ 的成员和友元以及 $Y2$ 的派生类（例如 $Z2$ ）的成员和友元在需要时可以（隐式地）将 $Y2^*$ 转换到 X^* ，就像它们可以访问 X 类的公用成员和保护成员一样。例如，

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // 可以: X是Y1的公用基类
    py1->a = 7;      // 可以

    px = py2;       // 可以: X是Y2的保护基类且Z2由Y2派生
    py2->a = 7;      // 可以

    px = py3;       // 错误: X是Y3的私用基类
    py3->a = 7;      // 错误
}
```

最后考虑

```
class Y3 : private X { void f(Y1*, Y2*, Y3*); };
```

因为 X 是 $Y3$ 的私用基类，只有 $Y3$ 的成员和友元在需要时可以（隐式地）将 $Y3^*$ 转换到 X^* ，就像它们可以访问 X 类的公用成员和保护成员一样。例如，

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
    X* px = py1;    // 可以: X是Y1的公用基类
    py1->a = 7;      // 可以

    px = py2;       // 错误: X是Y2的保护基类
    py2->a = 7;      // 错误

    px = py3;       // 可以: X是Y3的私用基类且Y3::f()是Y3的成员
    py3->a = 7;      // 可以
}
```

C.11.3 访问成员类

成员类的成员对于其外围类的成员并没有特殊的访问权。与此类似，外围类的成员对其嵌套类的成员也没有特殊的访问权，同样要遵守普通的访问规则（10.2.2节）。例如，

```
class Outer {
    typedef int T;
    int i;
public:
    int i2;
    static int s;

    class Inner {
        int x;
        T y; // 错误: Outer::T是私用的
    public:
        void f(Outer* p, int v);
    };

    int g(Inner* p);
};
```

```

};

void Outer::Inner::f(Outer* p, int v)
{
    p->i = v;      // 错误: Outer::i是私用的
    p->i2 = v;     // 可以: Outer::i2是公用的
}

int Outer::g(Inner* p)
{
    p->f(this, 2); // 可以: Inner::f()是公用的
    return p->x;   // 错误: Inner::x是私用的
}

```

然而, 授予成员类访问其外围类的权力通常是很有用的, 通过将成员类作为`friend`就可以做到这一点。例如,

```

class Outer {
    typedef int T;
    int i;
public:
    class Inner;      // 成员类的预先声明
    friend class Inner; // 将访问权授予Outer::Inner

    class Inner {
        int x;
        T y;        // 可以: Inner是友元
    public:
        void f(Outer* p, int v);
    };
};

void Outer::Inner::f(Outer* p, int v)
{
    p->i = v; // 可以: Inner是友元
}

```

C.11.4 友元关系

友元关系既不能继承, 也不能传递。例如,

```

class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p)
    {
        p->a++; // 错误: C不是A的友元, 虽然它是A的友元的友元
    }
};

class D : public B {
    void f(A* p)
    {

```

```

        p->a++; // 错误: D不是A的友元, 虽然它是由A的友元派生的
    }
};

```

C.12 到数据成员的指针

到成员的指针的概念 (15.5节) 理所当然地既适用于数据成员, 也适用于带有参数和返回值类型的成员函数。例如,

```

struct C {
    const char* val;
    int i;
    void print(int x) { cout << val << x << '\n'; }
    int f1(int);
    void f2();
    C(const char* v) { val = v; }
};

typedef void (C::*PMFI)(int); // 指向C的取一个int的成员函数
typedef const char* C::*PM;   // 指向C的char* 数据成员

void f(C& z1, C& z2)
{
    C* p = &z2;
    PMFI pf = &C::print;
    PM pm = &C::val;

    z1.print(1);
    (z1.*pf)(2);
    z1.*pm = "nv1 ";
    p->*pm = "nv2 ";
    z2.print(3);
    (p->*pf)(4);

    pf = &C::f1; // 错误: 返回类型不匹配
    pf = &C::f2; // 错误: 参数类型不匹配
    pm = &C::i;  // 错误: 类型不匹配
    pm = pf;     // 错误: 类型不匹配
}

```

对指向函数的指针类型的检查与其他类型完全一样。

C.13 模板

一个类模板描述了在给定一组合适的模板参数的情况下如何去生成一个类。与此类似, 一个函数模板描述了在给定一组合适的模板参数的情况下如何生成一个函数。这样, 模板就能用于生成类型和可执行代码。随着这种表达能力而来的是一些复杂性。这些复杂性中的大部分都与模板定义和使用的各种环境有关。

C.13.1 静态成员

类模板可以有 *static* 成员。由这个模板生成的每个类将有它自己的一份静态成员的副本。静态成员必须另行定义, 而且可以专门化。例如,

```

template<class T> class X {
    // ...

```

```

    static T def_val;
    static T* new_X(T a = def_val);
};

template<class T> T X<T>::def_val(0,0);
template<class T> T* X<T>::new_X(T a) { /* ... */ }

template<> int X<int>::def_val<int> = 0;
template<> int* X<int>::new_X<int>(int i) { /* ... */ }

```

如果你希望由一个模板生成的各个类中所有的成员能共享某个对象或者函数，你可以将它放到一个非参数化的基类里。例如，

```

struct B {
    static B* nil;    // 用在由B派生的每个类中，作为公共的空指针
};

template<class T> class X : public B {
    // ...
};

B* B::nil = 0;

```

C.13.2 友元

与其他类一样，模板类也可以有友元。考虑来自11.5节的`Matrix`和`Vector`的例子。通常`Matrix`和`Vector`这两个类都将是模板：

```

template<class T> class Matrix;

template<class T> class Vector {
    T v[4];
public:
    friend Vector operator*(<>(const Matrix<T>&, const Vector&);
    // ...
};

template<class T> class Matrix {
    Vector<T> v[4];
public:
    friend Vector<T> operator*(<>(const Matrix&, const Vector<T>&);
    // ...
};

```

在友元函数名字之后的`<>`是必须的，它将这个友元是模板的情况表述清楚了。如果没有`<>`，系统就会假定这是一个非模板函数。此后就可以写出直接访问`Matrix`和`Vector`的数据的乘法运算符了：

```

template<class T> Vector<T> operator* (const Matrix<T>& m, const Vector<T>& v)
{
    // ...用m.v[i] 和v.v[i] 直接访问元素...
}

```

友元并不影响模板类定义的作用域，也不影响模板使用的作用域。相反，对模板参数和运算符的查找是基于其参数进行的（11.2.3节、11.5.1节）。与成员函数一样，只有在调用时才会去实例化友元函数（C.13.9.1节）。

C.13.3 模板作为模板参数

将模板（而不是类或者对象）作为模板参数传递有时也很有用。例如：

```

template<class T, template<class> class C> class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};

Xrefd<Entry, vector> x1; // 将Entry的交叉引用存入vector
Xrefd<Record, set> x2;  // 将Record的交叉引用存入set

```

为了将一个模板声明为模板参数，我们就必须描述它所需要的参数。例如，我们在上面描述了 *Xrefd* 的模板参数 *C*，它就是有一个模板参数的模板类。如果不给以描述，我们就不能去使用 *C* 的专门化。采用模板作为模板参数，通常是我们希望用不同的参数对其进行实例化（例如上面例子中的 *T* 和 *T**），也就是说，我们希望借助于另一个模板来表达自己的模板里的成员声明，因此就需要以另一个模板作为参数，使用户可以描述它。

有一种常见情况，在这种情况下模板需要用一个容器来保存自己参数类型的元素，传递一个容器类型是处理这种问题的更好方式（13.6节、17.3.1节）。

只有类模板可以作为模板参数。

C.13.4 推断函数模板参数

如果模板函数的参数的类型是由下面各种结构组合而成的，编译器就可以推断出一个类型模板参数 *T* 或者 *TT*，以及一个非类型模板参数 *I*。

<i>T</i>	<i>const T</i>	<i>volatile T</i>
<i>T*</i>	<i>T&</i>	<i>T[constant_expression]</i>
<i>type[I]</i>	<i>class_template_name<T></i>	<i>class_template_name<I></i>
<i>TT<T></i>	<i>T<I></i>	<i>T<></i>
<i>T type::*</i>	<i>T T::*</i>	<i>type T::*</i>
<i>T (*) (args)</i>	<i>type (T::*) (args)</i>	<i>T (type::*) (args)</i>
<i>type (type::*) (args_TI)</i>	<i>T (T::*) (args_TI)</i>	<i>type (T::*) (args_TI)</i>
<i>T (type::*) (args_TI)</i>	<i>type (*) (args_TI)</i>	

在这里，*args_TI* 是一个参数表，通过递归地使用这些规则就可以从它确定得出 *T* 或者 *I*，而 *args* 是一个不允许推断的参数表。如果存在不能通过这种方式推断出的参数，这个调用就是有歧义的。例如，

```

template<class T, class U> void f(const T*, U(*) (U));

int g(int);

void h(const char* p)
{
    f(p, g); // T是char, U是int
    f(p, h); // 错误：无法推断U
}

```

先看 *f()* 的第一个调用的参数表，我们很容易就能推断出模板的各个参数。再看 *f()* 的第二个调用，我们可以看出 *h()* 不满足模式 *U(*) (U)*，因为 *h()* 的参数返回类型不同。

如果某个模板参数可以通过多个函数参数推断出来，那么所有这些推断都必须得到同一个类型，否则这个调用就是错误。例如，

```

template<class T> void f(T i, T* p);

void g(int i)

```



```
{
    f(i, &i);           // ok
    f(i, "Remember!"); // 错误, 歧义: T是int还是const char?
}
```

C.13.5 模板和typename

为了使通用型程序设计更容易并更具一般性, 标准库容器提供了一组标准函数和类型 (16.3.3节)。例如,

```
template<class T> class vector {
public:
    typedef T* iterator;
    iterator begin();
    iterator end();

    // ...
};

template<class T> class list {
    class link { /* ... */ };
public:
    typedef link* iterator;
    iterator begin();
    iterator end();

    // ...
};
```

这会引诱我们去写

```
template<class C> void f(C& v)
{
    C::iterator i = v.begin(); // 语法错误
    // ...
}
```

可惜编译器不是神仙, 所以它不知道`C::iterator`是个类型名。在某些情况中, 一个更聪明的编译器或许能猜测某个名字是有意要作为一个类型名, 或者是作为非类型的什么东西 (例如, 是个函数或者模板) 的名字, 但一般情况下这是不可能的。考虑下面这个专门剥去了有关其意义的线索的示例:

```
int y;

template<class T> void g(T& v)
{
    T::x(y); // 函数调用或变量声明?
}
```

`T::x`是一个以`y`作为参数的函数调用吗? 或者我们是想定义一个类型为`T::x`的局部变量`y`, 但乖张地多写了一对括号吗? 我们可以设想出一个环境, 使`X::x(y)`在其中是函数调用, 也可以设想出另一个环境使`Y::x(y)`是声明。

解决的方法很简单: 除非有另外的说明, 否则一个标识符总被假定引用的是某种非类型亦非模板的东西。如果我们想说某种东西应该被当做类型对待, 我们就可以借助于关键字 **typename** 做到这一点:

```
template<class C> void h(C& v)
{
    typename C::iterator i = v.begin();
    // ...
}
```

关键字`typename`也可以放在限定的名字之前,说明被命名的实体是一个类型。在这方面它就像`struct`或者`class`。

当某个类型名依赖于一个模板参数时,就需要用`typename`关键字了。例如,

```
template <class T> void k(vector<T>& v)
{
    vector<T>::iterator i = v.begin();           // 语法错误: 缺typename
    typename vector<T>::iterator i = v.begin(); // ok
    // ...
}
```

在这种情况下,编译器或许对每个`vector`的实例化都可能确定`iterator`是一个类型的名字,但没有要求编译器做到这件事。那样做是一种非标准的不可移植的语言扩展。只有在那些按语法只能允许类型名出现的情况下,编译器才会假定某个依赖于一个模板参数的名字是类型名。这类情况并不多见。例如,在`base-specifier`的位置(A.8.1节)。

`typename`还可以代替`class`用在模板声明中:

```
template<typename T> void f(T);
```

作为低水平的打字员,也为了节约屏幕空间,我喜欢采用更短的:

```
template<class T> void f(T);
```

C.13.6 `template`作为限定词

出现对`typename`限定词的需求,是因为我们既可以引用作为类型的成员,也可以引用非类型的成员。与此类似,也可能出现需要区分模板成员名与其他成员名的情况。考虑一个通用的存储管理器的可能界面:

```
class Memory { // 某个Allocator
public:
    template<class T> T* get_new();
    template<class T> void release(T&);
    // ...
};

template<class Allocator> void f(Allocator& m)
{
    int* p1 = m.get_new<int>();           // 语法错: 在小于后出现int
    int* p2 = m.template get_new<int>(); // 显式限定
    // ...
    m.release(p1); // 推断出模板参数,不必显式限定
    m.release(p2);
}
```

对`get_new()`必须做显式的限定,因为无法推断出它的模板参数。在这种情况下,必须使用前缀`template`去通知编译器(以及读程序的人),说明`get_new()`是一个成员模板,所以可以用所需要的元素类型去显式地限定。如果没有`template`限定词,我们就将得到一个编译错误,因为编译器会把`<`当做小于运算符。需要使用`template`限定的情况很少见,大部分模板参数都是

能够推断的。

C.13.7 实例化

有了一个模板定义和这个模板的一个使用，生成代码就是实现的工作了。从类模板和一组模板参数出发，编译器需要生成出一个类定义，并生成它的那些被使用了的成员函数的定义，对函数模板就需要生成一个函数。这一过程通常被称为模板的实例化。

那些生成出来的类和函数被称为专门化。在需要区分生成出来的专门化和由程序员写出的专门化（13.5节）时，也将它们分别称为生成的专门化和显式的专门化。显式的专门化有时也被称为用户定义的专门化，或者简称用户专门化。

为在非平凡的程序里使用模板，程序员必须理解模板定义中的名字是怎样与声明约束的，以及可能怎样去组织源代码（13.7节）。

按照默认方式，编译器依据名字约束规则（C.13.8节）从所用的模板出发去生成类和函数。也就是说，程序员不必显式地说明必须生成哪些模板的哪些版本。这件事非常重要，因为程序员很难知道究竟需要某个模板的哪些版本。经常有这种情况，许多程序员根本就没有听说过的模板被用在库的实现里，有时程序员所不知晓的模板使用的是他们同样并不知晓的模板参数类型。一般说，只有通过递归地检查应用代码库里所用的模板，才能弄清楚需要生成的函数集合。计算机比人更适合去做这种分析。

然而，对于程序员而言，能够去特别说明需要从某个模板生成代码这一点有时也很重要（C.13.10节）。通过这种工作，程序员能完成对实例化的详细控制。在大部分编译环境里，这也意味着确切地对做出哪些实例化进行控制。特别是，显式实例化还可以用于强迫编译错误在某个可预期的时刻发生，而不是在由实现确定的需要生成某个专门化的时刻发生。某些用户必须要求有一个完全可以预期的构造过程。

C.13.8 名字约束

在定义模板函数时有一件重要事情，那就是使它们尽可能少依赖于非局部的信息，因为模板将被用于去基于未知的类型、在未知的环境中生成函数和类。任何一点微妙的环境依赖性都可能在某个程序员排除程序错误时暴露出来，而程序员未必愿意了解模板的实现细节。尽可能避免全局名字的一般性规则，在模板代码中应该特别认真地考虑。由于这些，我们应该试着尽可能将模板定义做成自给自足的，把那些原本可能作为全局环境的东西以模板参数的方式提供给它（例如，特征类；13.4节、20.2.1节）。

然而，有时可能必须使用某些非局部名字。特别是人们常常写出一组相互合作的模板函数，而不是只写一个自足的函数。有时这些函数可以作为类的成员，但也未必总是如此。有时非局部函数是最好的选择，典型的例子是`sort()`调用`swap()`和`less()`（13.5.2节）。标准库算法可以看做一个大规模的示例（第18章）。

有着常规名字和语义的操作（例如`+`、`*`、`[]`和`sort()`）是模板定义中非局部名字的另一种来源。考虑

```
#include<vector>

bool tracing;

// ...
```

```

template<class T> T sum(std::vector<T>& v)
{
    T t = 0;
    if (tracing) cerr << "sum(" << &v << ")\n";
    for (int i = 0; i < v.size(); i++) t = t + v[i];
    return t;
}

// ...

#include<quad.h>

void f(std::vector<Quad>& v)
{
    Quad c = sum(v);
}

```

在这里，看起来很单纯的模板函数`sum()`依赖于`+`运算符。在这个例子里，`+`在`<quad.h>`里定义：

```
Quad operator+ (Quad, Quad);
```

重要的是，在定义`sum()`时作用域里没有任何东西与复数有关，也不能假定写`sum()`的人知道类`Quad`。特别是，这个`+`可能是在`sum()`之后的程序正文中定义，其定义的时间甚至也可能更晚。

为模板中显式或者隐式使用的每个名字确定声明的过程被称为名字约束。与模板名字约束有关的一般性问题就是模板实例化涉及到三个环境，而它们又无法清晰地区分：

- [1] 模板定义的环境。
- [2] 参数类型声明的环境。
- [3] 模板使用的环境。

C.13.8.1 依赖名

在定义一个模板函数时，我们希望能保证有足够的环境可用，使模板的定义在其实际参数的基础上取得自己的意义，而不会从使用点的环境中随便捡起一些“偶然的”内容。为了有助于达到这个目的，语言将模板定义中所使用的名字划分为两个类别：

- [1] 依赖于模板参数的名字。这种名字将在实例化的某一点被约束（C.13.8.4节）。在`sum()`的例子里，`+`的定义可以在实例化的环境中找到，因为它以模板参数类型为运算对象。
- [2] 不依赖于模板参数的名字。这种名字在模板的定义点被约束（C.13.8.2节）。在`sum()`的例子里，当编译器遇到`sum()`的定义时，模板`vector`已在标准头文件`<vector>`定义，布尔量`tracing`也在作用域中。

有关“*N*依赖于模板参数*T*”的最简单定义应该是“*N*是*T*的一个成员”。不幸的是，这样还不够；`Quad`例子（C.13.8节）里的加法就是一个反例。因此，说一个函数调用依赖于模板的某个参数，当且仅当下面条件之一成立：

- [1] 实际参数的类型按照类型推断规则（13.3.1节）依赖于模板参数*T*。例如，`f(T(l))`、`f(t)`、`f(g(t))`和`f(&t)`，假定*t*是*T*类型的。
- [2] 被调用函数的某个形式参数按照类型推断规则（13.3.1节）依赖于模板参数*T*。例如`f(T)`、`f(list<T>&)`和`f(const T*)`。

简而言之，一个被调用函数的名字有依赖，如果查看它的实际参数或者形式参数，可以明显

看到这种依赖性。

在调用中恰好有一个实际参数与某个实际模板参数匹配则不算是依赖。例如，

```
template<class T> T f(T a)
{
    return g(I);    // 错误: g() 不在作用域, g(I) 不依赖于T
}

int g(int);

int z = f(2);
```

对于调用`f(2)`, `T`正好是`int`而`g()`的实际参数正好也是`int`, 这个情况并无关系。如果将`g(I)`看做有依赖, 对于模板定义的读者而言, 它的意义就会过于微妙而神秘。如果程序员希望能调用`g(int)`, `g(int)`的声明就应该出现在`f()`的定义之前, 使得在分析`f()`时`g(int)`在作用域里。这也恰好与针对非模板函数定义的规则完全一样。

除了函数名之外, 变量、类型、`const`等的名字也可以是依赖的, 如果它们的类型依赖于模板参数。例如,

```
template<class T> void fct(const T& a)
{
    typename T::Memtype p = a.p;    // p和Memtype依赖于T
    cout << a.i << ' ' << p->j;    // i和j依赖于T
}
```

C.13.8.2 定义点约束

当编译器看到一个模板定义时, 它要确定哪些名字是依赖的 (C.13.8.1节)。如果某个名字是依赖的, 就将查找它的声明的工作推迟到实例化的时候再做 (C.13.8.3节)。

在函数的定义点, 所有不依赖于模板参数的名字都必须在作用域里 (4.9.4节)。例如,

```
int x;

template<class T> T f(T a)
{
    x++;    // ok
    y++;    // 错误: 没有y在作用域, 且y不依赖于T
    return a;
}

int y;

int z = f(2);
```

如果找到了一个声明, 那么就使用它, 即使是后来又遇到“更好的”声明。例如,

```
void g(double);

template<class T> class X : public T {
public:
    void f() { g(2); }    // 调用g(double)
    // ...
};

void g(int);

class Z { };

void h(X<Z> x)
{
```

```

    x.f();
}

```

在为 $X<Z>::f()$ 生成定义时，不会去考虑 $g(int)$ ，因为它是在 X 之后声明的，而 X 在 $g(int)$ 之后才使用的事实与此并无干系。还有，没有依赖关系的调用也不会在基类中遭到劫持：

```

class Y { public: void g(int); };

void h(X<Y> x)
{
    x.f();
}

```

$X<Y>::f()$ 还是会调用 $g(double)$ 。如果程序员原本希望能调用来自基类 T 的 $g()$ ，那就应该把 $f()$ 的定义写成

```

template<class T> class XX : public T {
    void f() { T::g(2); }    // 调用T::g()
    // ...
};

```

当然，这又是有关模板定义应该尽可能自足的经验规则的一个例证（C.13.8节）。

C.13.8.3 实例化点约束

某模板对一组给定模板参数的每一次使用都定义了一个实例化点。这个点位于包围这次使用的最近的全局的或者名字空间的作用域里，正好位于包含这次使用的那个声明之前。例如，

```

template<class T> void f(T a) { g(a); }

void g(int);

void h()
{
    extern g(double);
    f(2);
}

```

在这里， $f<int>()$ 的实例化点正好在 $h()$ 之前，因此，在 $f()$ 里所调用的 $g()$ 将是全局的 $g(int)$ ，而不是局部的 $g(double)$ 。“实例化点”的定义意味着模板参数绝不会约束于局部的名字或者类成员。例如，

```

void f()
{
    struct X { /* ... */ };    // 局部结构
    vector<X> v;                // 错误：不能使用局部结构作为模板参数
    // ...
}

```

在模板中使用的未加显式限定的名字也不会约束于局部名字。最后，即使某个模板已经先在一个类里使用了，模板里所用的未加限定的名字也不会约束于这个类的成员。我们必须这样忽略局部的名字，因为只有这样才能避免大量类似于宏的行为。例如，

```

template<class T> void sort(vector<T>& v)
{
    sort(v.begin(), v.end());    // 使用标准库sort()（没有显式地写std::）
}

class Container {
    vector<int> v;    // 元素
public:

```

```

void sort()    // 元素排序
{
    ::sort(v);    // sort(vector<int>&) 调用std::sort(), 而不是Container::sort()
}
// ...
};

```

如果将`sort(vector<T>&)`调用`sort()`用记法`std::sort()`写出, 其结果完全一样, 但代码却更清晰一些。

如果在一个名字空间里定义的某个模板的实例化点位于另一个名字空间里面, 来自两个名字空间的名字就都可以用于名字约束。与其他地方一样, 这时也通过重载解析, 从来自不同名字空间的名字中做出选择(8.2.9.2节)。

注意, 将同一组模板参数几次用于某个模板就出现了几个实例化点。如果其中独立名字的约束不同, 这个程序就是非法的。然而这是实现很难去检查的一种错误, 特别是如果这些实例化点位于不同的编译单位里。所以最好还是尽可能减少模板中非局部名字的使用, 并用头文件来保持环境的一致性, 以避免名字约束方面的这种难以捉摸的麻烦。

C.13.8.4 模板和名字空间

在调用某个函数时, 即使该函数不在作用域里也有可能找到它的声明, 只要函数的声明与其参数之一位于同一个名字空间里(8.2.6节)。对于在模板中调用的函数, 这一点非常重要, 因为这也就是在实例化过程期间找出依赖函数的机制。

模板的专门化可以在实例化的任何一点生成(C.13.8.3节)、在一个编译单位里实例化之后的任何一点生成, 或者在某个专门为生成一些专门化而创建的编译单位里生成。这也反应了为了生成专门化, 一个实现可以采用的三种明显策略:

- [1] 在第一次看到调用时生成对应的专门化。
- [2] 在一个编译单位的最后, 生成这个编译单位里所需的所有专门化。
- [3] 在看过一个程序的所有编译单位之后, 生成该程序所需要的所有专门化。

这三种策略各有各的强项和弱项, 也可能采用这些策略的组合方式。

在任何情况下, 独立名字的约束都在模板定义点完成, 完成依赖名的约束则需要查看:

- [1] 在模板定义点作用域里的那些名字。
- [2] 在一个依赖调用的参数的名字空间里的名字(全局函数在内部类型的名字空间里考虑)。

例如,

```

namespace N {
    class A { /* ... */ };
    char f(A);
}

char f(int);

template<class T> char g(T t) { return f(t); }

char c = g(N::A());    // 导致调用N::f(N::A)

```

这里的`f(t)`明显为依赖的, 所以我们就不能在定义点将`f`约束于`f(N::A)`或者`f(int)`。在为`g<N::A>(N::A)`生成专门化时, 实现将为`f()`去查看名字空间`N`并找到了`N::f(N::A)`。

在做专门化时, 如果通过选择不同的实例点, 或者在不同的可能环境中选择不同的名字空间内容, 有可能构造出两个不同的意义, 那么这个程序就是非法的。例如,

```

namespace N {
    class A { /* ... */ };
    char f(A, int);
}

template<class T, class T2> char g(T t, T2 t2) { return f(t, t2); }

char c = g(N::A(), 'a');    // 错误 (f(t) 存在多种解析)

namespace N {                // 加入名字空间N (8.2.9.3节)
    void f(A, char);
}

```

在这里，我们有可能在实例化点生成专门化，并由此使`f(N::A, int)`被调用。换个方式，我们也可能等待，并在编译单位的最后生成专门化，并使`f(N::A, char)`被调用。所以调用`g(N::A(), 'a')`是错误的。

在重载函数的两个声明之间调用它是一种很糟糕的程序设计。在查看一个大程序时，程序员没有道理去怀疑这种问题。对于上面这个特殊情况，编译器有可能捕捉到歧义性问题。但是，类似的问题可能出现在相互分离的不同编译单位中，检查它们就困难得多了。实现并没有责任去检查这种问题。

出现对函数调用的不同解析情况，大部分问题涉及到内部类型。因此，大部分修补都基于更加小心地使用内部类型的参数。

与其他地方一样，使用全局函数可能把问题变得更糟糕。全局名字空间被认为是与内部类型相关联的名字空间，因此，可以通过全局函数消解依赖于内部类型的调用。例如，

```

int f(int);

template<class T> T g(T t) { return f(t); }

char c = g('a');    // 错误：对f(t)存在多种解析

char f(char);

```

我们可能在实例化点生成专门化`g<char>(char)`，并使`f(int)`被调用。换个方式，我们也可能等到编译单位的最后生成专门化，并得到`f(char)`。所以调用`g('a')`是一个错误。

C.13.9 何时需要专门化

只有在需要某个类的定义时，才需要生成类模板的专门化。特别是，在声明对某个类的指针时，并不需要这个类的实际定义。例如，

```

class X;
X* p;    // 可以：不需要X的定义
X a;    // 错误：需要X的定义

```

在定义模板类时，这种区分可能是至关重要的。模板类将不被实例化，除非实际中需要它的定义。看下面例子：

```

template<class T> class Link {
    Link* suc; // 可以：(还)不需要Link的定义
    // ...
};

Link<int>* pl; // 不需要Link<int>的实例化
Link<int> lnk; // 现在需要实例化Link<int>了

```


实例化点就是第一次需要某个定义的地方。

C.13.9.1 模板函数实例化

只有在某个函数被使用时，实现才会去实例化它。特别是，实例化类模板并不意味着实例化它的所有成员，甚至不意味着实例化在模板类声明中定义的所有成员。这就使程序员在定义模板类时有了很大程度的灵活性。考虑

```
template<class T> class List {
    // ...
    void sort();
};

class Glob { /* 无比较运算符 */ };

void f(List<Glob>& lb, List<string>& ls)
{
    ls.sort();
    // 用lb上的操作，不包括lb.sort()
}
```

在这里，`List<string>::sort()` 被实例化，而 `List<Glob>::sort()` 并不实例化。这两方面都减少了所生成的代码量，也使我们不必重新设计这个程序。若是去生成 `List<Glob>::sort()` 的话，我们将不得不为 `Glob()` 增添 `List::sort()` 所需的操作，或是去重新定义 `sort()` 以使它不再是 `List` 的成员，或者是对 `Glob` 使用其他容器。

C.13.10 显式实例化

显式实例化请求也就是一个专门化声明，以 `template` 关键字开始（后面不跟 `<`）：

```
template class vector<int>;           // 类
template int& vector<int>::operator[] (int); // 成员
template int convert<int, double> (double); // 函数
```

模板声明以 `template<` 开始，而简单地写 `template` 则表示是实例化请求。请注意，以 `template` 开始的应该是一个完整声明，只说一个名字是不够的：

```
template vector<int>::operator[]; // 语法错
template convert<int, double>; // 语法错
```

与模板函数调用的情况一样，能够由函数参数推断出的模板参数都可以省略（13.3.1节）。例如，

```
template int convert<int, double> (double); // 可以（多余）
template int convert<int> (double); // 可以
```

当一个类模板被显式实例化时，它的所有成员函数都将实例化。

注意，显式实例化可以用做一种约束检查（13.6.2节）。例如，

```
template<class T> class Calls_foo {
    void constraints(T t) { foo(t); } // 从每个构造函数里调用
    // ...
};

template class Calls_foo<int>; // 错误: foo(int) 无定义
template void Calls_foo<Shape*>::constraints(Shape*); // 错误: foo(Shape*) 无定义
```

实例化请求有可能显著地提高连接和重新编译效率。我曾经看到这样的例子，通过将大部分模板实例化工作包装在一个编译单位里，编译的时间从若干小时减少到等值的若干分钟。

存在同一专门化的两个定义是一种错误,无论这种多重专门化是用户定义的(13.5节),是隐含生成的(C.13.7节),还是显式请求的。当然,并不要求编译器去检查不同编译单位里的多重专门化问题。这就使聪明的实现可以忽略多余的专门化,避免在用包含显式实例化的库(C.13.7节)组合出程序时可能出现的问题。当然,并不要求实现如此的聪明,“不够聪明”的实现的用户需要自己去避免多重实例化问题。当然,可能出现的最坏情况就是他们的程序无法装入,不会出现默默地改变意义的情况。

语言并不要求用户做显式的实例化。显式实例化只是一种为了优化和通过手工控制编译连接过程的可选机制(C.13.7节)。

C.14 忠告

- [1] 应集中关注软件开发而不是技术细节; C.1节。
- [2] 坚持标准并不能保证可移植性; C.2节。
- [3] 避免无定义行为(包括专有的扩充); C.2节。
- [4] 将那些实现定义的行为局部化; C.2节。
- [5] 在没有{、}、[、]、!或!的系统里用关键字和二联符表示程序,在没有\的地方用三联符; C.3.1节。
- [6] 为了方便通信,用ASCII字符去表示程序; C.3.3节。
- [7] 采用符号转义字符比用数值表示字符更好些; C.3.2节。
- [8] 不要依赖于`char`的有符号或者无符号性质; C.3.4节。
- [9] 如果对整数文字量的类型感到有疑问,请使用后缀; C.4节。
- [10] 避免破坏值的隐式转换; C.6节。
- [11] 用`vector`比数组好; C.7节。
- [12] 避免`union`; C.8.2节。
- [13] 用位域表示外部确定的布局; C.8.1节。
- [14] 注意不同存储管理风格间的权衡; C.9节。
- [15] 不要污染全局名字空间; C.10.1节。
- [16] 在需要作用域(模块)而不是类型的地方,用`namespace`比`class`更合适; C.10.3节。
- [17] 记住`static`类或员需要定义; C.13.1节。
- [18] 用`typename`消除对模板参数中类型成员的歧义性; C.13.5节。
- [19] 在需要用模板参数显式限定之处,用`template`消除模板类成员的歧义性; C.13.6节。
- [20] 写模板定义时,应尽可能减少对实例化环境的依赖性; C.13.8节。
- [21] 如果模板实例化花的时间过长,请考虑显式实例化; C.13.10节。
- [22] 如果需要编译顺序的显式可预见性,请考虑显式实例化; C.13.10节。

附录D 现 场

入乡随俗。

——成语

处理文化差异——类`locale`（现场）——命名的现场——现场的构建——现场的复制和比较——`global()`和`classic()`现场——比较字符串——类`facet`（刻面）——在现场里访问刻面——一个简单的用户定义刻面——标准刻面——字符串比较——数值I/O——货币I/O——日期和时间I/O——低级时间操作——`Date`类——字符类别——字符编码转换——消息分类——忠告——练习

D.1 处理文化差异

一个`locale`（现场）就是一个代表了一集文化习俗的对象，例如，字符串如何比较，供人阅读的数值的表现形式，在外部存储器里字符的表示形式等。现场的概念是可扩展的，所以，程序员可以向一个`locale`里加入新的`facet`（刻面），以便表示某些标准库没有直接支持的特殊实体，例如邮政编码和电话号码等。`locale`在标准库里的主要应用是控制送入`ostream`的信息的表现形式，以及从`istream`接受信息的格式。

21.7节描述了如何为一个流修改`locale`。本附录将描述`locale`是如何由`facet`构造出来的，并解释`locale`对流起作用的机制。本附录还将描述如何定义`facet`，列出一组定义流的特殊性质的标准`facet`，并介绍实现与使用各种`locale`和`facet`的一些技术。标准库表示日期和时间的功能将作为日期I/O的一部分予以讨论。有关现场和刻面的讨论的组织结构如下：

- D.1节 介绍利用`locale`表示文化差异的基本思想。
- D.2节 介绍`locale`类。
- D.3节 介绍`facet`类。
- D.4节 给出标准`facet`的概述，并介绍每个`facet`的细节。
 - D.4.1节 字符串比较。
 - D.4.2节 数值的输入和输出。
 - D.4.3节 货币值的输入和输出。
 - D.4.4节 日期和时间的输入和输出。
 - D.4.5节 字符类别。
 - D.4.6节 字符编码习惯。
 - D.4.7节 消息分类。

从本质上说，现场的概念并不是C++的概念。大部分操作系统和应用环境中都有现场的概念，在原则上，这种概念由一个系统里的所有程序所共享，并不依赖于它们用什么程序语言写出。这样，我们可以将C++现场这一标准库概念看做是一种标准的可移植的方式，C++程序借助它去访问不同系统中在表示方式上差异很大的信息。特别是，C++的`locale`是与系统信息之间的

一个公共界面，而在不同的系统里这些信息可能具有互不兼容的表示方式。

D.1.1 对文化差异编程

考虑编写一个需要在几个国家里使用的程序。以这种允许在多个国家使用的风格写程序常常被称为“国际化”（强调程序在多个国家使用）或者“本地化”（强调程序能够适应本地的情况）。在不同的国家里，由程序操作的许多实体需要按不同的习惯显示出来，只要在写I/O过程时考虑有关的需求，我们就可以处理这些习惯了。例如，

```
void print_date(const Date& d) // 按适当格式打印
{
    switch(where_am_I) {      // 用户定义的格式指示符
    case DK:                   // 例如, 7. marts 1999
        cout << d.day() << ". " << dk_month[d.month()] << " " << d.year();
        break;
    case UK:                   // 例如, 7/3/1999
        cout << d.day() << " / " << d.month() << " / " << d.year();
        break;
    case US:                   // 例如, 3/7/1999
        cout << d.month() << "/" << d.day() << "/" << d.year();
        break;
    // ...
    }
}
```

这种风格的代码能完成有关工作。但是，它相当难看，而且我们必须一致地使用这种风格，以保证所有的输出都能正确地适应本地习惯。更糟糕的是，如果需要增加一种新的日期书写方式，我们就必须修改代码。我们可以设想通过创建一个类层次结构来处理这个问题（12.2.4节）。但是，位于一个**Date**里的信息与我们希望看它的方式无关。因此，我们希望的并不是**Date**类型的层次结构，如**US_date**、**UK_date**和**JP_date**等；相反，我们需要的是显示**Date**的各种不同方式、例如美国风格的输出、英国风格的输出和日本风格的输出等。见D.4.4.5节。

由“让用户去写关注文化差异的I/O函数”还会引起另外一些问题：

- [1] 应用程序员在没有标准库帮助的情况下，不能很容易地、可移植地、有效地修改内部类型的表现形式。
- [2] 在大程序里找出每一个I/O操作（以及以某种对现场敏感的方式为I/O准备数据的每一个操作）并不总是可行的。
- [3] 有时我们无法重写程序来处理一套新的习惯，即使能做，我们也更希望有无须重写的解决方案。
- [4] 让每个用户为不同文化差异的问题设计和实现一个解决方案太浪费了。
- [5] 不同的程序员将会以不同方式去处理低层次的文化习俗，这将使程序由于并非基本的原因，以不同方式去处理相同的信息。这样，维护来自不同地方的代码的程序员就必须学习各种编程习惯，这是令人厌烦的，也很容易出错。

由于这些原因，标准库提供了一种处理文化习惯的可扩展的方式。**iostream**库（21.7节）依赖于此框架来处理内部的和用户定义的类型。举个例子，考虑复制 (**Date**, **double**) 对偶的一个简单循环，它表示的可能是一系列测量值或者一组事务：

```
void cpy(istream& is, ostream& os) // 复制 (Date, double) 流
{
    Date d;
    double volume;

    while (is >> d >> volume) os << d << " " << volume << "\n";
}
```

当然，实际程序还会对这些记录做一些事，理想情况下还需要小心地处理错误。

我们怎么能让一个程序去读入符合法语习惯的文件（其中在浮点数里用逗号作为表示小数点的符号，例如12,5表示12又二分之一），并让它按照美国的习惯输出呢？我们可以定义 *locale* 和 I/O 操作，使 *cpy()* 能够用于习惯间的转换：

```
void f(istream& fin, ostream& fout, istream& fin2, ostream& fout2)
{
    fin.imbue(locale("en_US")); // 美国英语
    fout.imbue(locale("fr")); // 法语
    cpy(fin, fout); // 读美国英语、写法语

    fin2.imbue(locale("fr")); // 法语
    fout2.imbue(locale("en_US")); // 美国英语
    cpy(fin2, fout2); // 读法语，写美国英语
}
```

对于流

```
Apr 12, 1999 1000.3
Apr 13, 1999 345.45
Apr 14, 1999 9688.321
...

3 juillet 1950 10,3
3 juillet 1951 134,45
3 juillet 1952 67,9
...
```

这个程序将产生

```
12 avril 1999 1000,3
13 avril 1999 345,45
14 avril 1999 9688,321
...

July 3, 1950 10.3
July 3, 1951 134.45
July 3, 1952 67.9
...
```

本附录中剩下的大部分内容都是有关如何完成这些工作的机制的描述，并解释如何去使用它们。请注意，大部分程序员很少有理由去处理 *locale* 的细节，他们永远也不会显式地去操纵某个 *locale*，其中大部分人要做的只是提取出一个标准的现场，用它去浸染（*imbue*）一个流（21.7节）。当然，用于组合出这些 *locale* 并使它们易于使用的有关机制本身也组成了很小的程序设计语言。

如果一个程序或者系统很成功，它就会被那些抱有原设计师和程序员未曾预料的需要和喜好的人们所用。最成功的程序将在一些国家里运行，那里所用的自然语言和字符集与原设计师和程序员所熟悉的完全不同。程序的广泛使用是成功的标志，所以，为跨越语言的和文

化的边界而设计和编程，也就是为成功做准备。

本地化（和国际化）的概念很简单，然而，各种现实约束却使*locale*的设计和实现变得相当复杂：

- [1] 一个*locale*要封装起许多文化习惯，例如日期的表现形式。这些习惯变化多端，有许多细微之处和不系统性。这些习惯与程序设计语言没有任何关系，因此程序设计语言无法去标准化它们。
- [2] *locale*的概念必须是可扩展的，因为不可能列举出对每个C++用户具有重要意义的每一种文化习惯。
- [3] *locale*被用在I/O操作中，在这里人们非常注重运行效率。
- [4] *locale*必须对大部分程序员是不可见的，因为大部分人所需要的不过是流I/O能“做正确的事情”，并不关心所做的究竟是什么，也不关心它们如何能做到这些。
- [5] *locale*必须能为那些涉及各种与文化有关的信息处理功能的人所用，不应局限在流I/O库的范围之内。

设计一个做I/O的程序需要做出一个选择，是通过“传统代码”去控制格式呢，还是用*locale*。如果我们能保证每一个输入操作都能很容易地从一种约定形式转换到另一种约定形式，那么前一种方式（传统方式）是可行的。在另一方面，如果内部类型的表现形式需要变化，如果需要不同的字符集合，或者如果我们需要在一组可扩展的I/O约定形式中做出选择，那么*locale*机制就很有吸引力了。

一个*locale*由一些*facet*组成，这些*facet*控制着各个方面，比如在输出浮点数时小数点所用的字符（*decimal_point*()；D.4.2节），读入货币值时所用的格式（*money_punct*；D.4.3节）。*facet*是类*locale::facet*（D.3节）的某个派生类的对象，我们也可以将*locale*看做是*facet*的容器（D.2节、D.3.1节）。

D.2 类*locale*

*locale*类以及与之相关的功能在<*locale*>里给出：

```
class std::locale {
public:
    class facet;           // 用于表示locale各个方面的类型；D.3节
    class id;              // 用于标识locale的类型；D.3节
    typedef int category;  // 用于对facet分组/划分类别的类型

    static const category // 由实现定义的实际值
        none = 0,
        collate = 1,
        ctype = 1<<1,
        monetary = 1<<2,
        numeric = 1<<3,
        time = 1<<4,
        messages = 1<<5,
        all = collate | ctype | monetary | numeric | time | messages;

    locale() throw();      // 全局locale的副本（D.2.1节）
    locale(const locale& x) throw(); // x的副本
    explicit locale(const char* p); // 名字为p的locale的副本（D.2.1节）

    ~locale() throw();
```

```

locale(const locale& x, const char* p, category c); // x的副本加上来自p中类别c的facet
locale(const locale& x, const locale& y, category c); // x的副本加上来自y中类别c的facet

template <class Facet> locale(const locale& x, Facet* f); // x的副本加上facet f
template <class Facet> locale combine(const locale& x); // this* 的副本加上来自x的facet

const locale& operator=(const locale& x) throw();

bool operator==(const locale&) const; // 比较locale
bool operator!=(const locale&) const;

string name() const; // 本locale的名字 (D.2.1节)

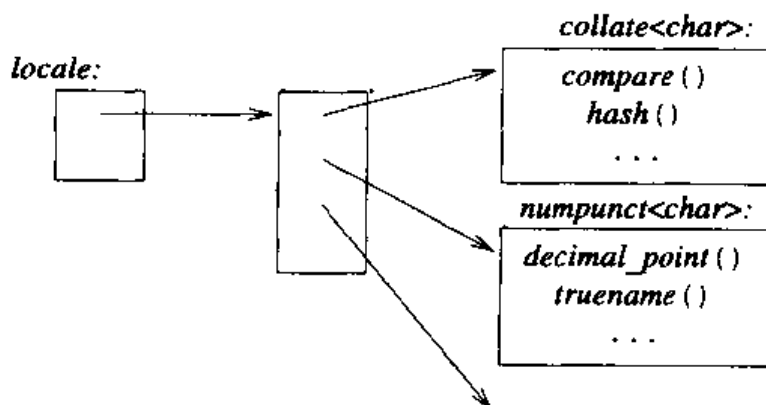
template <class Ch, class Tr, class A> // 用这个locale比较字符串
bool operator()(const basic_string<Ch, Tr, A>&, const basic_string<Ch, Tr, A>&) const;

static locale global(const locale&); // 设置全局locale, 返回原locale
static const locale& classic(); // 取得“经典的”C风格的locale

private:
    // 表示
};

```

一个**locale**可以看做是到一个`map<id, facet*>`的界面；也就是说，它是某种使我们可以用一个**locale::id**去找出与之对应的、由**locale::facet**派生的类的一个对象的东西。**locale**的实际实现就是这个思想的一种高效率的变形。其结构布局大致类似下面的样子：



这里的**collate<char>**和**numpunct<char>**都是标准库**facet** (D.4节)。与所有**facet**一样，它们也是由**locale::facet**派生出来的。

locale的复制应该是自由的、代价低廉的。因此，**locale**几乎可以肯定被实现为某种句柄，指向包含了它的主要实现部分的特定`map<id, facet*>`。**locale**里的**facet**必须能够快速访问，因此，这个特殊的`map<id, facet*>`应该被优化，以提供某种类似数组的快速访问方式。对某个**locale**里的**facet**的访问采用`use_facet<Facet>(loc)`的记法形式；见D.3.1节。

标准库提供了很丰富的一组**facet**。为帮助程序员按照逻辑分组操作**facet**，标准**facet**被分成一些类别，例如**numeric**和**collate** (D.4节)。

程序员可以替换现存类别中的**facet** (D.4节、D.4.2.1节)，但却不能加入新的类别，程序员没有办法定义新的类别。“类别”的概念只适用于标准库剖面中，而且它是不可扩展的。一个剖面可以不属于任何类别，许多用户定义的剖面都是这样。

对**locale**的最大多数使用都是在流I/O里的隐式使用。每个**istream**和**ostream**都有自己的**locale**。在创建流时，流的**locale**默认为是全局**locale** (D.2.1节)。流的**locale**可以通过**imbue()**操作设置，而且我们可以用**getloc()**操作提取某个流的**locale**的一个副本 (21.6.3节)。

D.2.1 命名的现场

用一个`locale`和一些`facet`可以创建一个新的`locale`。创建`locale`的最简单方式就是复制某个已有的`locale`。例如，

```
locale loc0; // 复制当前的全局现场 (D.2.3节)
locale loc1 = locale(); // 复制当前的全局现场 (D.2.3节)
locale loc2(""); // 复制“用户偏爱的现场”
locale loc3("C"); // 复制"C"现场
locale loc4 = locale::classic(); // 复制"C"现场
locale loc5("POSIX"); // 复制实现所定义的"POSIX"现场
```

标准把`locale("C")`的意义定义为“经典的”C现场，这也就是在本书中自始至终所用的现场。其他`locale`的名字由实现定义。

`locale("")`被称做“用户偏爱的现场”，这个现场将在程序的执行环境里由语言之外设定某种意义。

大部分操作系统都有为程序设定现场的能力。通常情况下，适用于所用系统的某个个人的现场是在该个人首次登录系统时选择的。例如，我可以期望，某个将系统配置为以阿根廷的西班牙语为默认设置的人将会发现`locale("")`就是`locale("es_AR")`。对我所用系统做快速检查，发现了51个具有助记名的现场，例如`POSIX`、`de`、`en_UK`、`en_US`、`es`、`es_AR`、`fr`、`da`、`pl`、`iso_8859_1`。`POSIX`建议了一种现场命名格式：采用小写字母的语言名，后跟可选的大写的国家名，后面再跟可选的编码描述符；例如`jp_JPjit`。但无论如何，对于这些名字并没有跨平台标准。在其他系统里，在许多其他`locale`名字之中，我还看到了`g`、`uk`、`us`、`fr`、`sw`、`da`等。`C++`标准并不为任何确定的国家或语言的`locale`定义某种意义，虽然可能存在某个平台的特殊标准。因此，要使用在某个系统里命名的`locale`，程序员必须去查阅系统手册并做试验。

一个具有普遍性的好办法就是避免将`locale`的名字嵌入程序的正文之中。在程序正文里直接提到一个文件或者一个系统的名字将会限制该程序的可移植性，并迫使想把这个程序调整到另一个新环境的程序员去查找并修改这些值。直接提到现场名字字符串也有类似的恼人后果。与此相反，现场的名字应该由程序执行环境中得到（例如，采用`locale("")`），或者程序员可以询问专家用户，要他们通过输入字符串描述所需要的现场。例如，

```
void user_set_locale(const string& question_string)
{
    cout << question_string; // 例如，“如果你希望采用其他现场，请输入它的名字”
    string s;
    cin >> s;
    locale::global(locale(s.c_str())); // 将用户给定的现场设置为全局现场
}
```

更好的方式是让那些非专家用户从一组选项中挑选。完成此事的例程需要知道系统将自己的现场保存在哪里，以及它们是如何保存的。

如果字符串参数并没有引用某个已定义的`locale`，构造函数就抛出`runtime_error`异常（14.10节）。例如，

```
void set_loc(locale& loc, const char* name)
try
{
```



```

    loc = locale(name);
}
catch (runtime_error) {
    cerr << "locale \"" << name << "\" isn't defined\n";
    // ...
}

```

如果一个`locale`有名字字符串，`name()`将返回它。如果没有，`name()`将返回`string("")`。名字字符串主要是作为一种引用存储在执行环境中的`locale`的方式。其次，名字字符串也可以作为一种排错的辅助手段。例如，

```

void print_locale_names(const locale& my_loc)
{
    cout << "name of current global locale: " << locale().name() << "\n";
    cout << "name of classic C locale: " << locale::classic().name() << "\n";
    cout << "name of ``user's preferred locale``: " << locale("").name() << "\n";
    cout << "name of my locale: " << my_loc.name() << "\n";
}

```

如果两个现场具有相同的名字，而且名字不同于`string("")`，比较时将认为它们是相等的。当然，`==`和`!=`为比较`locale`提供了更为直接的方式。

通过名字字符串取得的某个`locale`的副本具有与原`locale`同样的名字（如果原`locale`有名字的话），所以很可能有许多`locale`的名字相同。这也符合逻辑，因为`locale`是不变的，所有这些同名对象定义的都是同一组文化习惯。

调用`locale(loc, "Foo", cat)`将做出一个现场，它很像`loc`，除了取得了`locale("Foo")`中类别`cat`的那些刻面之外。当且仅当`loc`有名字串时，这个结果现场也有一个名字串。标准并没有明确说出新现场的名字串是什么，但假定了它与`loc`的名字串不同。一种明显的实现方式是用`loc`的名字字符串和`"Foo"`组合出新名字。例如，假定`loc`的名字是`en_UK`，新现场有可能以`"en_UK:Foo"`作为它的名字字符串。

新建`locale`的名字字符串可以总结如下：

现 场	名字字符串
<code>locale("Foo")</code>	<code>"Foo"</code>
<code>locale(loc)</code>	<code>loc.name()</code>
<code>locale(loc, "Foo", cat)</code>	新串，如果 <code>loc</code> 有名字串；否则是 <code>string("")</code>
<code>locale(loc, loc2, cat)</code>	新串，如果 <code>loc</code> 和 <code>loc2</code> 有名字串；否则是 <code>string("")</code>
<code>locale(loc, Facet)</code>	<code>string("")</code>
<code>loc.combine(loc2)</code>	<code>string("")</code>

并不存在一种机制使程序员可以给定一个C风格字符串作为在系统里新建立的`locale`的名字。名字字符串或者是由系统的执行环境定义的，或者是由这种名字再经过`locale`的构造函数组合而成的。

D.2.1.1 创建新现场

做出新现场的方式就是取一个现存的`locale`，添加或者替换一些`facet`。一个典型的新`locale`都是某个现存物经过略微变形的结果。例如，

```

void f(const locale& loc, const My_money_io* mio) // My_money_io在D.4.3.1节定义
{

```

```

    locale loc1(locale("POSIX"), loc, locale::monetary); // 用loc的货币facet
    locale loc2 = locale(locale::classic(), mio); // classic加上mio
    // ...
}

```

这里的`loc1`是`POSIX`现场的一个副本，被修改为使用`loc`的货币`facet`（D.4.3节）。与此类似，`loc2`是C现场的副本，修改为使用`My_money_io`（D.4.3节）。如果`Facet*`（这里是`My_money_io`）参数是0，结果`locale`就是函数的`locale`参数的一个副本。

如果用

```
locale(const locale& x, Facet* f);
```

那么参数`f`必须能标明某个特定的刻面类型，一个普通的`facet*`是不够的。例如：

```

void g(const locale::facet* mio1, const My_money_io* mio2)
{
    locale loc3 = locale(locale::classic(), mio1); // 错误: facet类型未知
    locale loc4 = locale(locale::classic(), mio2); // 可以: facet类型已知
    // ...
}

```

这里的原因是，`locale`在编译时需要用`Facet*`参数的类型去确定刻面的类型。特别是，`locale`的实现需要利用刻面的标识类型`facet::id`（D.3节）在现场里找到对应的刻面（D.3.1节）。

注意，构造函数

```
template <class Facet> locale(const locale& x, Facet* f);
```

是语言中提供的惟一机制，使程序员能通过`locale`提供一个可用的`facet`。其他`locale`都是由实现者以命名的`locale`的方式提供的（D.2.1节）。这些命名的`locale`可以由程序执行环境中获取。如果程序员了解在这里所用的由实现确定的机制，那就可以按照同样方式加入新的`locale`（D.6[11, 12]）。

`locale`的构造函数集合的设计使得每个`facet`的类型都是已知的，或者可以通过类型推断得知（从`Facet`模板参数），或者是因为它来自其他的`locale`（因此类型已知）。明确描述了`category`参数也就间接地确定了`facet`的类型，因为`locale`知道这个类别中的`facet`的类型。这也意味着`locale`类能够（而且也确实）维持着有关`facet`类型的信息，以便它能以最小的开销去操纵它们。

`locale::id`成员类型被`locale`用来标识`facet`的类型（D.3节）；

从某`locale`构造出一个现场，这个现场的一个`facet`来自另外一个`locale`，这种方式有时也很有用。`combine()`模板成员函数做的就是这件事。例如，

```

void f(const locale& loc, const locale& loc2)
{
    locale loc3 = loc.combine<My_money_io>(loc2);
    // ...
}

```

结果`loc3`的行为类似于`loc`，除了它采用来自`loc2`的`My_money_io`（D.4.3.1节）做货币的格式化I/O。如果`loc2`里不存在需要提供给新`locale`的`My_money_io`，`combine()`将抛出一个`runtime_error`异常（14.10节）。`combine()`的结果没有名字串。

D.2.2 现场的复制和比较

现场可以通过初始化或者赋值的方式复制。例如：

```

void swap(locale& x, locale& y)    // 就像std::swap()
{
    locale temp = x;
    x = y;
    y = temp;
}

```

`locale`的副本与原本比较起来是相等的，但副本是另一个独立的对象。例如，

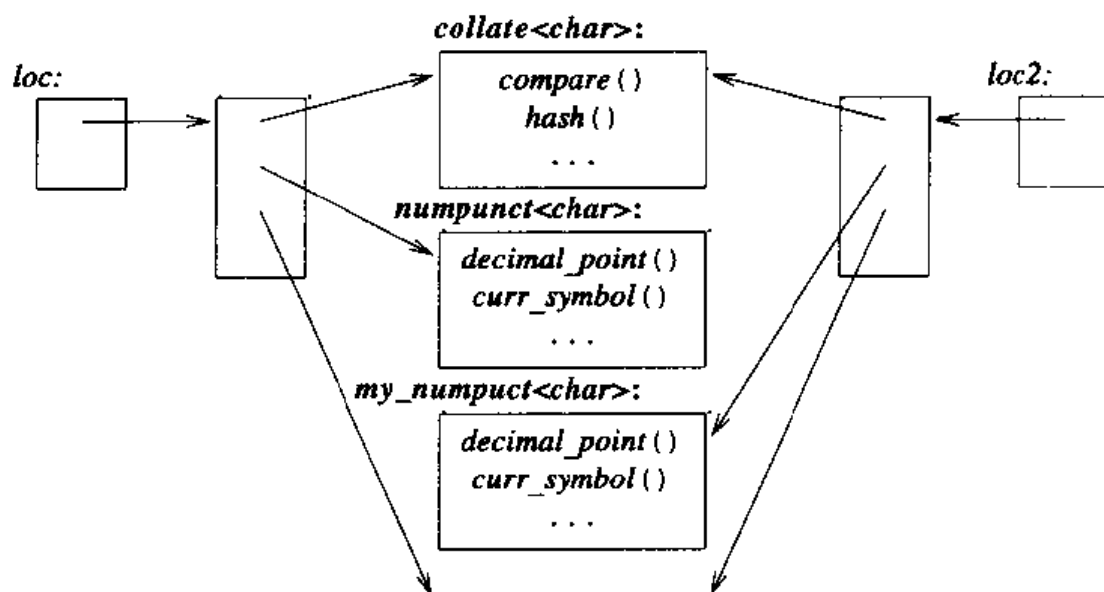
```

void f(locale* my_locale)
{
    locale loc = locale::classic(); // "C" 现场
    if (loc != locale::classic()) {
        cerr << "implementation error: send bug report to vendor\n";
        exit(1);
    }

    if (&loc != &locale::classic()) cout << "no surprise: addresses differ\n";
    locale loc2 = locale(loc, my_locale, locale::numeric);
    if (loc == loc2) {
        cout << "my numeric facets are the same as classic()'s numeric facets\n";
        // ...
    }
    // ...
}

```

如果现场`my_locale`有一个数值标点刻画`my_numpunct<char>`，它与`classic()`的标准`numpunct<char>`不同，结果的现场可以表示为下面的形式：



并不存在修改`locale`的手段，所有`locale`操作都是从现存的`locale`做出新的`locale`。`locale`在创建之后就不能修改，这一点对于运行效率至关重要。因为这就使人们可以去调用`facet`的虚函数，并将返回的值缓存起来。例如，一个`istream`可以知道用于表示小数点的字符和`true`的表示方式，就不必在每次读数时都去调用`decimal_point()`，也不必每次读`bool`时都去调用`truename()`（D.4.2节）。只有对流调用`imbue()`才会使对这些函数的调用返回不同的值。

D.2.3 `global()` 和 `classic()` 现场

一个程序的当前现场的概念由 `locale()` 和 `locale::global(x)` 提供, 前者产生出当前现场的一个副本, 后者将当前现场设置为 `x`。当前现场也常被称为“全局现场”, 这也反应了它可能被实现为一个全局的 (或者 `static`) 对象。

一个流在初始化时将隐式地使用全局现场。也就是说, 每个新的流都用 `locale()` 的副本浸染 (21.1节、21.6.3节)。初始时的全局现场就是标准的C现场 `locale::classic()`。

静态函数 `locale::global()` 使程序员可以将某个现场指定为全局现场。 `global()` 返回前一全局现场的一个副本, 这就使用户可以恢复原来的全局现场。例如,

```
void f(const locale& my_loc)
{
    ifstream fin1(some_name);           // fin1被全局现场浸染
    locale& old_global = locale::global(my_loc); // 设置新的全局现场
    ifstream fin2(some_other_name);     // fin2用my_loc浸染
    // ...
    locale::global(old_global);          // 恢复老的全局现场
}
```

如果现场 `x` 有名字字符串, `locale::global(x)` 还将同时设置C全局现场。这也就意味着, 如果一个C++程序调用了来自C标准库的与现场有关的函数, 在整个混合式的C++和C程序中对现场的处理将是一致的。

如果现场 `x` 没有名字字符串, `locale::global(x)` 是否影响C全局现场就没有定义了。这也意味着C++程序将不能可靠而可移植地将C现场设置为某个无法由执行环境提取出的现场。不能从C程序里设置C++全局现场 (除非是调用C++函数去完成此事)。在混合式的C++和C程序里, 让C全局现场与 `global()` 不同很容易引起错误。

设置全局现场不会影响现存的I/O流, 它们将仍旧使用在全局现场重新设置之前浸染了它们的那个现场。例如, 通过操作全局现场使 `fin2` 被 `my_loc` 浸染, 将不会影响到 `fin1`。

就像所有依赖于修改全局数据的技术一样, 修改全局现场也会遇到一些麻烦: 本质上说, 无法知道这样的一个修改将影响到哪些东西。因此, 应当尽可能减少 `global()` 的使用, 并将这种改变局部到若干遵照简单的修改策略的代码段里。用特定 `locale` 浸染单个流 (21.6.3节) 的能力使这件事情很容易做到。当然, 显式地在程序里到处使用 `locale` 和 `facet` 也会变成维护中的问题。

D.2.4 字符串比较

依据一个 `locale` 去比较两个字符串可能是 `locale` 最常见的显式使用。因此, `locale` 直接提供了这个操作, 这就使用户不必再基于其 `collate` 刻画 (D.4.1节) 去构造自己的比较函数了。为能直接作为谓词 (18.4.2节) 使用, 这个比较函数被定义为 `locale` 的 `operator()()`。例如,

```
void f(vector<string>& v, const locale& my_locale)
{
    sort(v.begin(), v.end());           // 排序中用 < 比较元素
    // ...
    sort(v.begin(), v.end(), my_locale); // 排序中依照my_locale的规则
    // ...
}
```

按照默认规定，标准库的`sort()`用实现字符集所对应数值的`<`去确定检查顺序（18.7节、18.6.3.1节）。

注意，`locale`比较的是`basic_string`，而不是C风格的字符串。

D.3 刻画

一个刻画也就是`locale`的成员类`facet`的某个派生类的一个对象：

```
class std::locale::facet {
protected:
    explicit facet(size_t r = 0);    // “r == 0”：让现场控制这个刻面的生存时间
    virtual ~facet();               // 注意：保护析构函数
private:
    facet(const facet&);              // 不定义
    void operator=(const facet&);    // 不定义

    // 表示
};
```

复制操作是`private`且不予定义以阻止复制（11.2.2节）。

`facet`的设计就是作为一个基类，它也没有公用函数。它的构造函数是`protected`，以防创建“一般的`facet`”对象；其析构函数是虚函数，以保证能正确析构派生类的对象。

设计`facet`的意图就是由`locale`通过指针去管理它们。如果`facet`构造函数的参数为0，意味着`locale`在对这个`facet`的最后一个引用离开时应该删除它，相反，非零构造函数参数保证`locale`从不删除`facet`。非零参数很少使用，用于表示该`facet`的生存时间由程序员直接控制，而不是间接地通过`locale`。举例来说，我们可能希望像下面这样创建一些标准刻画类型`collate_byname<char>`（D.4.1.1节）的对象：

```
void f(const string& s1, const string& s2)
{
    // 正常情况：（默认）参数0表示现场负责删除：
    collate<char>* p = new collate_byname<char>("pl");
    locale loc(locale(), p);

    // 罕见情况：参数1表示用户负责删除：
    collate<char>* q = new collate_byname<char>("ge", 1);

    collate_byname<char> bug1("sw");    // 错误：不能删除局部变量
    collate_byname<char> bug2("no", 1); // 错误：不能删除局部变量

    // ...

    // q不能删除；collate_byname<char>的析构函数是保护的
    // 没有删除p；现场负责删除 *p
}
```

也就是说，在由现场或者基类管理，或偶然地用其他方式管理下，各种标准刻画都很有用。

`_byname()`刻画就是来自执行环境中的某个命名的现场的刻画（D.2.1节）。

要在一个`locale`里找`facet`，可以通过`has_facet()`和`use_facet()`（D.3.1节），每一类`facet`都必须有一个`id`：

```
class std::locale::id {
public:
    id();
private:
```

```

    id(const id&);           // 不定义
    void operator=(const id&); // 不定义

    // 表示
};

```

复制操作被声明为私用，而且不予定义以阻止复制（11.2.2节）。

预想中`id`的使用方式是让用户对每个支持新`facet`界面的类（例如，见D.4.1节）定义一个类型为`id`的`static`成员。`locale`机制通过`id`去标识`facet`（D.2、D.3.1节）。在`locale`的明显实现方式中，`id`用于作为到刻面的指针向量的下标，进而实现一种高效率的`map<id, facet*>`。

用于定义某个（派生的）`facet`的数据将在派生类里定义，而不是在基类`facet`自身中定义。这就意味着，程序员定义的`facet`对于数据有完全的控制，可以使用任意数量的数据去表示该`facet`所代表的那个概念。

注意，一个用户定义`facet`的所有成员函数都必须是`const`成员。一般说，`facet`的本意就是不可改变的（D.2.2节）。

D.3.1 访问一个现场里的刻面

对一个`locale`里的`facet`的访问通过模板函数`use_facet`完成，我们也可以用模板函数`has_facet`询问在一个`locale`里是否存在某个特定的`facet`：

```

template <class Facet> bool has_facet(const locale&) throw();
template <class Facet> const Facet& use_facet(const locale&); // 可能抛出bad_cast

```

可以将这些模板函数设想为在它们的`locale`参数里查找它们的模板参数`Facet`。换一种方式，也可以把`use_facet`想像为一类从一个`locale`到某种特定`facet`的显式类型转换（强制）。这也是可行的，因为在一个`locale`里，特定类型的`facet`只有一个。例如，

```

void f(const locale& my_locale)
{
    char c = use_facet< numpunct<char> >(my_locale).decimal_point() // 使用标准facet
    // ...

    if (has_facet<Encrypt>(my_locale)) { // my_locale里包含Encrypt刻面吗？
        const Encrypt& f = use_facet<Encrypt>(my_locale); // 提取Encrypt刻面
        const Crypto c = f.get_crypto(); // 使用Encrypt刻面
        // ...
    }
    // ...
}

```

注意，`use_facet`返回到一个到`const`刻面的引用，因此我们不能将`use_facet`的结果赋值给一个非`const`。这也是有意义的，因为刻面本意就是不变的，只包含`const`成员。

如果我们试图做`use_facet<X>(loc)`，而在这个`loc`里并没有`X`刻面，`use_facet`就会抛出`bad_cast`异常（14.10节）。对于所有现场，标准`facet`都保证可以使用（D.4节），所以我们不必对标准刻面去用`has_facet`。对于标准刻面，`use_facet`也不会抛出`bad_cast`异常。

`use_facet`和`has_facet`可能如何实现呢？应该记得，我们可以将`locale`看做是一个`map<id, facet*>`（D.2节）。将某个刻面类型作为`Facet`模板参数，`use_facet`和`has_facet`的实现就可以引用到`Facet::id`，并采用它去找出对应的刻面。`use_facet`和`has_facet`的一种朴素实现看起来可能是下面的样子：

```
// 伪实现：设想现场有一个名为facet_map的map<id, facet*>

template <class Facet> bool has_facet(const locale& loc) throw()
{
    const locale::facet* f = loc.facet_map[Facet::id];
    return f ? true : false;
}

template <class Facet> const Facet& use_facet(const locale& loc)
{
    const locale::facet* f = loc.facet_map[Facet::id];
    if (f) return static_cast<const Facet&>(*f);
    throw bad_cast();
}
```

看*Facet::id*机制的另一种方式是将其看成一种编译时多态性的实现。可以用*dynamic_cast*得到类似于*use_facet*所产生的结果。当然，与一般性的*dynamic_cast*相比，更特殊的*use_facet*的实现有可能做得效率更高一些。

*id*实际标识的是一个界面及其相应行为，而不是一个类。也就是说，如果两个*facet*类具有完全相同的界面并实现了相同的语义（在*locale*所关心的范围内），它们就应该由同样的*id*所标识。例如，在一个*locale*里的*collate<char>*和*collate_byname<char>*可以互换，所以两者都由*collate<char>::id*所标识（D.4.1节）。

如果我们定义一个具有新界面的*facet*（例如*f()*里的*Encrypt*），我们就必须定义一个对应的*id*去标识它（见D.3.2节、D.4.1节）。

D.3.2 一个简单的用户定义剖面

标准库为文化差异的大部分关键领域提供了各种标准剖面，例如对字符集和数值I/O。广泛使用的类型比较复杂，需要特别关注与之相关的效率。为远离这些情况，以便孤立地考察剖面机制，让我们首先介绍一个针对简单的用户定义类型的*facet*：

```
enum Season { spring, summer, fall, winter };
```

这几乎是我可以想到的最简单的用户定义类型。在这里所勾勒出的I/O风格可以用于大部分用户定义类型，或许要做一点改动：

```
class Season_io : public locale::facet {
public:
    Season_io(int i = 0) : locale::facet(i) {}
    ~Season_io() {} // 使可能销毁Season_io对象（D.3节）
    virtual const string& to_str(Season s) const = 0; // s的字符串表示
    // 将对应于s的Season放入x
    virtual bool from_str(const string& s, Season& x) const = 0;
    static locale::id id; // 剖面标识符对象（D.2节、D.3节、D.3.1节）
};

locale::id Season_io::id; // 定义剖面标识符对象
```

为了简单起见，将这个*facet*限制在只使用*char*作为表示。

*Season_io*类为所有*Season_io*剖面提供了一种具有一般性的抽象界面。为针对某个特定*locale*定义*Season*的I/O表示形式，我们需要从*Season_io*派生出一个类，并适当地定义*to_str()*

和`from_str()`。

`Season`的输出很容易。如果某个流有一个`Season_io`剖面，我们就可以用它将值转换成字符串。如果它没有，我们就可以输出`Season`的`int`值：

```
ostream& operator<<(ostream& s, Season x)
{
    const locale& loc = s.getloc(); // 提取流的现场 (21.7.1节)
    if (has_facet<Season_io>(loc)) return s << use_facet<Season_io>(loc).to_str(x);
    return s << int(x);
}
```

注意，这个 `<<` 运算符是通过调用其他类型的 `<<` 实现的。我们采用这种方式可以得到许多利益：比直接访问`ostream`的流缓冲区简单得多，利用了这些 `<<` 运算符对现场的敏感性，利用了这些 `<<` 运算符所提供的错误处理。为了最大限度地提供效率和灵活性，标准`facet`倾向于直接对流缓冲区操作（D.4.2.2、D.4.2.3节）。但是对于大多数用户定义类型而言，完全没有必要钻到`streambuf`抽象层去。

与其他情况一样，输入的处理比输出更复杂一些：

```
istream& operator>>(istream& s, Season& x)
{
    const locale& loc = s.getloc(); // 提取流的现场 (21.7.1节)
    if (has_facet<Season_io>(loc)) { // 读入字母表示
        const Season_io& f = use_facet<Season_io>(loc);
        string buf;
        if (! (s>>buf && f.from_str(buf, x)) ) s.setstate(ios_base::failbit);
        return s;
    }
    int i; // 读入数值表示
    s>>i;
    x = Season(i);
    return s;
}
```

错误处理非常简单，也按照对内部类型的错误处理风格。就是说，如果输入串并不表示所用现场的一个`Season`，流将被置入`fail`状态。如果启用异常，这就意味着要抛出一个`ios_base::failure`异常（21.3.6节）。

下面是一个简单的测试程序：

```
int main() // 简单测试
{
    Season x;
    // 使用默认现场（没有Season_io剖面）隐含着整数I/O；
    cin >> x;
    cout << x << endl;

    locale loc(locale(), new US_season_io);
    cout.imbue(loc); // 用有Season_io剖面的现场
    cin.imbue(loc); // 用有Season_io剖面的现场

    cin >> x;
    cout << x << endl;
}
```

给了输入

2

summer

程序的响应是

2

summer

为了得到这些，我们必须定义*US_season_io*来定义表示季节的字符串表示，并覆盖那些在字符串表示与枚举符之间转换的*Season_io*函数：

```
class US_season_io : public Season_io {
    static const string seasons[];
public:
    const string& to_str(Season) const;
    bool from_str(const string&, Season&) const;
    // 注意：没有US_season_io
};

const string US_season_io::seasons[] = { "spring", "summer", "fall", "winter" };

const string& US_season_io::to_str(Season x) const
{
    if (s<spring || winter<s) {
        static const string ss = "no-such-season";
        return ss;
    }
    return seasons[x];
}

bool US_season_io::from_str(const string& s, Season& x) const
{
    const string* beg = &seasons[spring];
    const string* end = &seasons[winter]+1;
    const string* p = find(beg, end, s); // 3.8.1节、18.5.2节
    if (p==end) return false;
    x = Season(p-beg);
    return true;
}
```

注意，因为*US_season_io*只是*Season_io*界面的一个实现，我没有为*US_season_io*定义一个*id*。事实上，如果希望将*US_season_io*用做*Season_io*，我们就绝不能为*US_season_io*定义它自己的*id*。有关*locale*的操作，例如*has_facet*（D.3.1节），都依赖于一个事实：实现同样概念所有刻面都由同一个*id*标识（D.3节）。

最有趣的实现问题是，遇到要求输出一个非法的*Season*时应该怎么办。很自然，这根本就不应该出现，但是，遇到一个简单用户定义类型的非法值也是常见的情况，因此将这种情况考虑在内也非常实际。我也可以采用抛出异常的方式，但是，在处理简单的意在给人看的输出时，更有帮助的是对超出范围的值产生一个“超出范围”的表示形式。注意，对输入而言，错误处理策略是将这种情况留给 `>>` 运算符；而对于输出，这里是让刻面函数*to_str*()实现一种错误处理策略。这样做的目的就是为了显示不同的设计选择。在“产品设计”中，*facet*函数应该同时为输入和输出实现错误处理机制，或者将错误都报告给 `>>` 和 `<<` 去处理。

这个*Season_io*的设计依赖于派生类来提供与现场有关的字符串。另一种设计可能是让*Season_io*自己去从一个由现场确定的储藏库里找出这些字符串（D.4.7节）。还有一种可能性：

只用一个`Season_io`类，将特定的季节字符串作为它的构造函数的参数，这一方式留做练习(D.6[2])。

D.3.3 现场和刻面的使用

在标准库里，*locale*的主要使用是在I/O流中。然而，*locale*机制是一种用于表示对文化敏感的信息的具有普遍性和可扩展性的机制。*message* 剖面(D.4.7节)就是一个完全与I/O流无关的剖面实例。对于*iostream*库的扩充，甚至一些非基于流的I/O功能，都有可能利用*locale*的优点。还有，用户也可以用*locale*作为自己组织对文化敏感的信息的一种方便方式。

因为*locale/facet*机制的通用性，用户定义*facet*具有无限的可能性。可能的*facet*候选者包括日期、时区、电话号码、社会保障号(个人识别号)、产品编码、温度、一般的(单位, 值)对、邮政编码、服装大小和ISBN编码等的表示。

与其他功能强大的机制一样，对*facet*的使用也应该小心。某些东西可以表示为*facet*并不意味着它最好就采用这样的表示。为某种文化依赖性选择表示方式时，需要考虑的关键问题是各种决策：对于写代码的困难程度、结果代码的易读程度、结果程序的可维护性以及结果I/O操作在时间与空间方面的效率等方面的影响。

D.4 标准剖面

在`<locale>`里，标准库为*classic()*现场提供了如下*facet*：

标准剖面 (在 <i>classic()</i> 现场)			
	类 别	用 途	刻 面
D.4.1节	<i>collate</i>	字符串比较	<i>collate</i> <Ch>
D.4.2节	<i>numeric</i>	数值I/O	<i>num_punct</i> <Ch> <i>num_get</i> <Ch> <i>num_put</i> <Ch>
D.4.3节	<i>monetary</i>	货币I/O	<i>moneypunct</i> <Ch> <i>moneypunct</i> <Ch, true> <i>money_get</i> <Ch> <i>money_put</i> <Ch>
D.4.4节	<i>time</i>	时间I/O	<i>time_get</i> <Ch> <i>time_put</i> <Ch>
D.4.5节	<i>ctype</i>	字符分类	<i>ctype</i> <Ch> <i>codecvt</i> <Ch, char, mbstate_t>
D.4.7节	<i>messages</i>	消息提取	<i>messages</i> <Ch>

在这个表里，*Ch*是*char*或者*wchar_t*的缩写。需要用标准I/O处理其他的字符类型*X*的用户必须为*X*提供适当的*facet*版本。例如，可能需要用*codecvt*<*X*, *char*, *mbstate_t*>(D.4.6节)去控制在*X*和*char*之间的转换。这里的*mbstate_t*类型用于表明一种多字节字符表示的移位状态(D.4.6节)，*mbstate_t*类型在*<wchar>*和*<wchar_t.h>*里定义。对于任意字符类型*X*，与*mbstate_t*等价的是*char_traits*<*X*>::*state_type*。

此外，标准库还在`<locale>`提供了下述各*facet*：

标准剖面			
	类别	用途	剖面
D.4.1节	<i>collate</i>	字符串比较	<i>collate_byname</i> <Ch>
D.4.2节	<i>numeric</i>	数值I/O	<i>num_punct_byname</i> <Ch> <i>num_get</i> <C, In> <i>num_put</i> <C, Out>
D.4.3节	<i>monetary</i>	货币I/O	<i>moneypunct_byname</i> <Ch, International> <i>money_get</i> <C, In> <i>money_put</i> <C, Out>
D.4.4节	<i>time</i>	时间I/O	<i>time_put_byname</i> <Ch, Out>
D.4.5节	<i>ctype</i>	字符分类	<i>ctype_byname</i> <Ch>
D.4.7节	<i>messages</i>	消息提取	<i>messages_byname</i> <Ch>

在实例化这个表里的某个*facet*时，*Ch*可以是*char*或者*wchar_t*，*C*可以是任何字符类型（20.1节）。*international*可以为*true*或者*false*；*true*表示当前字符采用一种4字符的“国际”表示形式（D.4.3.1节）。*mbstate_t*类型用于表明一种多字节字符表示的移位状态（D.4.6节），*mbstate_t*类型在<*cwchar*>和<*wchar_t.h*>里定义。

*In*和*Out*分别是输入和输出迭代器（19.1节、19.2.1节）。为*_get*和*_put*剖面提供了这些模板参数就使程序员能够通过*facet*去访问非标准的缓冲区（D.4.2.2节）。与*iostream*相关联的缓冲区是流缓冲区，所以，为它们提供的迭代器是*ostreambuf_iterator*（19.2.6.1节、D.4.2.2节）。因此，错误处理就可以利用函数*failed()*（19.2.6.1节）。

一个*F_byname*剖面是由*F*剖面派生的。*F_byname*剖面提供了与*F*一样的界面，除此之外它还增加了一个构造函数，其参数是一个命名*locale*的字符串（见D.4.1节）。*F_byname(name)*为定义在*locale(name)*里的*F*提供了适当的语义。这里的想法是从程序执行环境里的一个命名的*locale*（D.2.1节）中取出一个标准剖面的版本。例如，

```
void f(vector<string>& v, const locale& loc)
{
    locale dl(loc, new collate_byname<char>("da")); // 用丹麦字符串比较
    locale dk(dl, new ctype_byname<char>("da"));    // 用丹麦字符分类
    sort(v.begin(), v.end(), dk);
    // ...
}
```

这个新的*dk*现场将使用丹麦风格的字符串，但保留数值的默认约定。注意，因为*facet*的第二个参数默认是0，这个*locale*将负责管理在这里由运算符*new*创建的*facet*的生存时间（D.3节）。

与以字符串为参数的*locale*类似，*_byname*的构造函数将访问程序的执行环境，这也意味着它们比那些无须参考环境的构造函数慢得多。首先创建起一个现场，而后访问它的各个剖面，比在程序里的许多地方使用*_byname*剖面要快得多。也就是说，只从环境里读一次剖面，而后反复使用位于内存中的副本通常都是一种好办法。例如：

```
locale dk("da"); // 读入丹麦现场（包括其所有剖面）一次
                // 而后根据需要使用dk现场及其剖面

void f(vector<string>& v, const locale& loc)
{
    const collate<char>& col = use_facet<collate<char>>>(dk);
    const collate<char>& ctyp = use_facet<ctype<char>>>(dk);
```

```

    locale d1(loc, col); // 使用丹麦字符串比较
    locale d2(d1, ctype); // 使用丹麦字符分类和丹麦字符串比较

    sort(v.begin(), v.end(), d2);
    // ...
};

```

类别的概念提供了一种更简单的操纵现场中标准刻面的方式。例如，有了`dk`现场，我们可以创建一个`locale`，让它按照丹麦语的规则（其中有着比英语更多的元音）比较字符串，但继续使用C++的数值语法形式：

```

    locale dk_us(locale::classic(), dk, collate|ctype); // 丹麦语字母，美国数值

```

对各个标准`facet`的介绍中包含了更多有关`facet`使用的示例。特别是，关于`collate`的讨论（D.4.1节）展示了剖面中许多公共结构的情况。

注意，不同的标准`facet`一般都是相互依赖的。例如，`num_put`依赖于`num_punct`。你只有了解了有关各个独立刻面的细节知识之后，才能成功地混合和匹配使用这些标准剖面，或者向其中加入新版本。换句话说，除了在21.7节中介绍的那些简单操作之外，`locale`并不是一种希望新手直接使用的机制。

各个剖面的设计常常极为繁琐。究其原因，部分是因为剖面必须去反应那些设计师无法控制的繁琐的文化习惯，部分是因为C++的标准库功能需要尽可能地维持与C标准库和各种平台的特定标准的兼容性。例如，POSIX提供了现场功能，库的设计师如果忽略它们将是不明智的。

另一方面，由现场和剖面方式提供的这个框架是充分一般而且非常灵活的。可以设计出能保存任意数据的剖面，可以用剖面操作提供所需要的对这些数据的任何操作。如果某个新剖面的行为并没有受到习俗的过度约束，其设计将会是简单而清晰的（D.3.2节）。

D.4.1 字符串比较

标准`collate`剖面提供了比较`Ch`类型的字符数组的各种方式：

```

template <class Ch>
class std::collate : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit collate(size_t r = 0);

    int compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const
        { return do_compare(b, e, b2, e2); }

    long hash(const Ch* b, const Ch* e) const { return do_hash(b, e); }
    string_type transform(const Ch* b, const Ch* e) const { return do_transform(b, e); }

    static locale::id id; // 剖面标识符对象（D.2节、D.3节、D.3.1节）

protected:
    ~collate(); // 注意：保护的析构函数

    virtual int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    virtual string_type do_transform(const Ch* b, const Ch* e) const;
    virtual long do_hash(const Ch* b, const Ch* e) const;
};

```

与所有刻画一样，*collate*也由*facet*公用派生，也提供了一个构造函数，该构造函数用一个参数告知是否要求*locale*类控制这个刻面的生存期间（D.3节）。

注意，这里的析构函数是保护的，*collate*刻面的意图并不是直接使用，而是想作为所有（派生的）整理（collation）类的基类，使*locale*能够管理它们（D.3节）。应用程序员、实现的提供者和库的提供商将写出各种字符串比较刻画，并让人们通过*collate*提供的界面去使用它们。

*compare()*函数依据由特定*collate*定义出的规则完成基本的字符串比较；如果第一个串按照字典顺序大于第二个时它将返回1，两个串相等时返回0，第二个串大于第一个时返回-1。例如，

```
void f(const string& s1, const string& s2, collate<char>& cmp)
{
    const char* cs1 = s1.data(); // 因为compare()在char[]上操作
    const char* cs2 = s2.data();
    switch (cmp.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size())) {
        case 0: // 按照cmp规则相等的串
            // ...
            break;
        case -1: // s1 < s2
            // ...
            break;
        case 1: // s1 > s2
            // ...
            break;
    }
}
```

请注意，*collate*的成员函数比较的是*Ch*的数组而不是*basic_string*或者以0结束的C风格字符串。特别是，这里将数值为0的*Ch*当做普通字符，而不是作为结束符。还有，*compare()*与*strcmp()*不同，它返回的是确切的-1、0、1值，不像*strcmp()*返回0和（任意的）正负值（20.4.1节）。

标准库的*string*对*locale*并不敏感，也就是说，它总是依据实现字符串（C.2节）的规则去做字符串比较。进一步说，标准*string*也没有提供描述比较准则的直接方式（第20章）。为了完成对*locale*敏感的比较，我们可以采用有关的*collate*的*compare()*。从记述形式看，通过*locale*的*operte()*间接使用*collate*的*compare()*更方便些。例如，

```
void f(const string& s1, const string& s2, const char* n)
{
    bool b = s1 == s2; // 用实现的字符集值做比较

    const char* cs1 = s1.data(); // 因为compare()在char[]上操作
    const char* cs2 = s2.data();

    typedef collate<char> Col;

    const Col& glob = use_facet<Col>(locale()); // 取自当前全局现场
    int i0 = glob.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());

    const Col& my_coll = use_facet<Col>(locale("")); // 取自我所喜欢的现场
    int i1 = my_coll.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());

    const Col& coll = use_facet<Col>(locale(n)); // 取自命名的现场n
    int i2 = coll.compare(cs1, cs1+s1.size(), cs2, cs2+s2.size());
}
```

```

    int i3 = locale()(s1, s2);           // 按当前全局现场比较
    int i4 = locale("")(s1, s2);        // 按我所喜欢的现场比较
    int i5 = locale(n)(s1, s2);          // 按命名的现场n比较
    // ...
}

```

在这里有 $i0 == i3$, $i1 == i4$, $i2 == i5$ 。不难设想出一些使 $i2$ 、 $i3$ 、 $i4$ 互不相同的情况。考虑下面来自德语字典的一系列单词:

Dialekt, Diät, dich, dichten, Dichtung

按照德语习惯, (只有) 名词采用大写, 而顺序与大小写无关。

如果按大小写敏感方式做德语单词排序, 就会把所有 **D** 开头的词放在 **d** 开头的词之前:

Dialekt, Diät, Dichtung, dich, dichten

这里的 *ä* (变音的 *a*) 应处理为 “一种 *a*”, 因此它应该出现在 *c* 之前。然而, 在大部分常见字符集里, *ä* 的数值都大于 *c* 的数值, 因此就有 $\text{int}('c') < \text{int}('ä')$, 基于数值的简单的默认排序将产生出:

Dialekt, Dichtung, Diät, dich, dichten

写一个能按字典给出正确序列的比较函数是一个很有趣的练习 (D.6[3])。

函数 `hash()` 计算出一个散列值 (17.6.2.3 节), 显然这对于构造散列表很有用处。

函数 `transform()` 产生出一个字符串, 用其他字符串与它比较, 这个字符串给出的将是与原参数字符串比较的同样结果。`transform()` 的意图是产生出某种优化代码, 用于做一个字符串与许多字符串的比较。这种功能对于在一组字符串里检索一个或者几个字符串很有价值。

公用的 `compare()`、`hash()` 和 `transform()` 函数的实现方法就是分别去调用保护的虚函数 `do_compare()`、`do_hash()` 和 `do_transform()`。这些 “do_函数” 可以由派生类覆盖。这种双函数策略使写非虚函数的库实现者能为所有调用提供某种公共的功能, 使之能独立于由用户提供的 `do_函数` 所做的事情。

虚函数的使用使 `facet` 具有多态性, 但也要付出执行的代价。为了避免更多的函数调用, `locale` 可以确定所用的到底是哪个 `facet`, 并缓存起为高效执行所需要的任何值 (D.2.2 节)。

类型为 `locale::id` 的静态成员 `id` 用于标识一个 `facet` (D.3 节)。标准函数 `has_facet` 和 `use_facet` 依赖于 `id` 和 `facet` 之间的对应关系 (D.3.1 节), 为 `locale` 提供同样界面和语义的两个 `facet` 应该有同一个 `id`。举例说, `collate<char>` 和 `collate_byname<char>` (D.4.1.1 节) 具有同样的 `id`。相反的, (就 `locale` 的角度看) 执行不同功能的两个 `facet` 必须具有不同的 `id`。例如, `num_punct<char>` 和 `num_put<char>` (D.4.2 节) 的 `id` 就不同。

D.4.1.1 命名的 `collate`

一个 `collate_byname` 是一个刻面, 它根据由其构造函数的字符串参数确定名字的现场, 提供一个 `collate` 版本:

```

template <class Ch>
class std::collate_byname : public collate<Ch> {
public:
    typedef basic_string<Ch> string_type;

    explicit collate_byname(const char*, size_t r = 0); // 由命名现场创建

    // 注意: 没有 id, 没有新函数

```

```
protected:
    ~collate_byname(); // 注意：保护的析构函数
    // 覆盖collate<Ch>的虚函数：
    int do_compare(const Ch* b, const Ch* e, const Ch* b2, const Ch* e2) const;
    string_type do_transform(const Ch* b, const Ch* e) const;
    long do_hash(const Ch* b, const Ch* e) const;
};
```

这样，`collate_byname`就可以用于在程序的执行环境中通过一个现场名字取出一个`collate` (D.4节)。在执行环境里存储刻面的最明显方式是将其作为数据存入文件。另一种不那么灵活的方式是将刻面表示为`_byname`刻面里的程序正文和数据。

`collate_byname<char>`类是没有自己`id`的`facet`的一个例子 (D.3节)。在一个`locale`里，`collate_byname<Ch>`和`collate<Ch>`是可以互换的。对于同一个现场，`collate_byname`和`collate`的差异只在于`collate_byname`多提供了一个构造函数，以及`collate_byname`所提供的语义。

注意，`_byname`的析构函数是保护的，这就意味着你不能将`_byname`刻面作为局部变量。例如：

```
void f()
{
    collate_byname<char> my_coll(""); // 错误：无法销毁my_coll
    // ...
}
```

这反应了一种观点：对于现场和刻面的使用最好是在程序的某个相当高的层次中做，去影响比较大的代码体。有关的例子如设置全局现场 (D.2.3节) 或者浸染一个流 (21.6.3节、D.1节)。如果有必要，我们也可以从一个`_byname`类派生出一个带公用析构函数的类，并创建该类的局部变量。

D.4.2 数值的输入和输出

数值输出是由`num_put`刻面向流缓冲区写 (21.6.4节) 来完成的，对应的数值输入是由`num_get`刻面从流缓冲区读来完成的。`num_put`和`num_get`所用的格式是由“数值标点”刻面`num_punct`定义的。

D.4.2.1 数值标点

`num_punct`刻面定义各种内部类型的I/O格式，例如`bool`、`int`和`double`等：

```
template <class Ch>
class std::num_punct : public locale::facet {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit num_punct(size_t r = 0);

    Ch decimal_point() const;           // classic()里是 '.'
    Ch thousands_sep() const;           // classic()里是 ','
    string grouping() const;             // classic()里是 ""；表示不分组

    string_type truename() const;        // classic()里是 "true"
    string_type falsename() const;       // classic()里是 "false"

    static locale::id id; // 刻面标识符对象 (D.2节、D.3节、D.3.1节)
```

```
protected:
    ~numpunct();

    // 为公用函数用的“do_”虚函数（见D.4.1节）
};
```

`grouping()` 的返回串里的字符被当做一系列小的整数值。每个数表示一组的数字个数。第0个字符表示最右的一组（最低位数字），第1个字符表示它左边的那一组，以次类推。按这种规定，“\004\002\003”就描述了一个数，例如123-45-6789（如果你用‘-’作为分隔字符的话）。如果实际需要，分组模式的最后一个数就将被反复使用，所以“\003”等价于“\003\003\003”。分隔字符就像其名字 `thousands_sep()` 所表示的，分组的最主要用途是为改善大数的可读性。`grouping()` 和 `thousands_sep()` 函数定义了一种格式，用于完成整数的输入和输出。它们并不被用于浮点数的I/O。这样，如果我们只定义 `grouping()` 和 `thousands_sep()`，不可能使1234567.89被打印成1,234,567.89。

我们可以通过由 `numpunct` 派生定义出新的标点风格。例如，我可以定义类 `My_punct`，用空格分开3个数字为一组的方式写整数，浮点数采用欧洲风格，用逗号作为“小数点”：

```
class My_punct : public std::numpunct<char> {
public:
    typedef char char_type;
    typedef string string_type;

    explicit My_punct(size_t r = 0) : std::numpunct<char>(r) {}

protected:
    char do_decimal_point() const { return ','; } // 逗号
    char do_thousands_sep() const { return ' '; } // 空格
    string do_grouping() const { return "\003"; } // 3个一组
};

void f()
{
    cout << "style A: " << 12345678 << " *** " << 1234567.8 << '\n';

    locale loc(locale(), new My_punct);
    cout.imbue(loc);
    cout << "style B: " << 12345678 << " *** " << 1234567.8 << '\n';
}
```

这将产生

```
style A: 12345678***1.23457e+06
style B: 12 345 678***1,23456e+06
```

注意，`imbue()` 将其参数的一个副本存入它的流里，因此流可以依赖于这个浸染的现场，即使原来那个现场已经被销毁了。如果某个 `iostream` 设置了自己的 `boolalpha` 标志（21.2.2节、21.4.1节），由 `truenamex()` 和 `falsenamex()` 返回的字符串将分别用于表示 `true` 和 `false`；如果没有设置就用1和0。

提供了一个 `_byname` 版本的 `numpunct`（D.4节、D.4.1节）：

```
template <class Ch>
class std::numpunct_byname : public numpunct<Ch> { /* ... */ };
```

D.4.2.2 数值的输出

在向流缓冲区写的过程中（21.6.4节），`ostream` 将依靠 `num_put` 类：


```

template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::num_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit num_put(size_t r = 0);

    // 将值v放入流s中的由b定位的缓冲区:
    Out put(Out b, ios_base& s, Ch fill, bool v) const;
    Out put(Out b, ios_base& s, Ch fill, long v) const;
    Out put(Out b, ios_base& s, Ch fill, unsigned long v) const;
    Out put(Out b, ios_base& s, Ch fill, double v) const;
    Out put(Out b, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, ios_base& s, Ch fill, const void* v) const;

    static locale::id id; // 刻面标识符对象 (D.2节、D.3节、D.3.1节)

protected:
    ~num_put();

    // 为公用函数用的“do_”虚函数 (见D.4.1节)
};

```

输出迭代器 (19.1节、19.2.1节) 参数 *Out* 标明了 *ostream* 流缓冲区里 (21.6.4节) 的一个位置, *put()* 把被输出数值的字符表示放到那里。*put()* 的值是一个迭代器, 定位在所写的最后字符之后的位置。

请注意 *num_put* 的默认专门化 (迭代器将通过它去访问字符, 其类型为 *ostreambuf_iterator<Ch>*), 它是标准现场的一部分 (D.4节)。如果你希望采用另外的专门化, 就必须自己将它写出来。例如,

```

template<class Ch>
class String_numput : public std::num_put<Ch, typename basic_string<Ch>::iterator> {
public:
    String_numput() : std::num_put<Ch, typename basic_string<Ch>::iterator>(1) {}
};

void f(int i, string& s, int pos) // 将i的格式化结果放入s从pos开始的位置
{
    String_numput<char> f;
    ios_base& xxx = cout; // 用cout的格式化规则
    f.put(s.begin()+pos, xxx, ' ', i); // 将i的格式化结果放入s
}

```

ios_base 参数用于取得有关格式化状态和现场的信息。例如, 如果需要填充, 所用的填充字符 *fill* 将由 *ios_base* 参数那里获取。在典型情况下, 通过 *b* 写入的流缓冲区也就是以 *s* 作为基类的 *ostream* 所关联的缓冲区。注意, 一个 *ios_base* 并不是一个能简单地创建的对象。特别是它控制着格式化的许多方面, 为了得到可接受的输出, 这些方面就必须保持协调一致。正是由于这些情况, *ios_base* 没有公用构造函数 (21.3.3节)。

put() 函数也使用自己的 *ios_base* 参数取得流的 *locale()*。这个 *locale* 被用于确定标点 (D.4.2.1节)、布尔量的字母表示形式以及到 *Ch* 的转换方式。例如, 假定 *s* 是 *put()* 的 *ios_base* 参数, 我们可能在 *put()* 函数里找到类似下面这样的代码:

```

const locale& loc = s.getloc();
// ...
wchar_t w = use_facet<ctype<char>>>(loc).widen(c); // char到Ch的转换

```

```
// ...
string pnt = use_facet< numput<char> > (loc) .decimal_point(); // 默认: '.'
// ...
string flse = use_facet< numput<char> > (loc) .falsename(); // 默认: "false"
```

像`num_put<char>`这样的标准刻画通常都是在标准I/O流函数里隐式地使用的,因此大部分程序员都不必知道它们。然而,标准库函数对这些刻画的使用还是很有趣的,因为它们显示了I/O流如何工作以及刻画如何使用。与其他地方一样,在这里,标准库也提供了许多有趣的程序设计技术示例。

利用`num_put`, `ostream`的实现者可能写出

```
template<class Ch, class Tr>
ostream& std::basic_ostream<Ch, Tr>::operator<< (double d)
{
    sentry guard(*this); // 见21.3.8节
    if (!guard) return *this;
    try {
        if (use_facet< num_put<Ch> > (getloc())) .put(*this,*this,this ->fill(), d) .failed()
            setstate(badbit);
    }
    catch (...) {
        handle_ioexception(*this);
    }
    return *this;
}
```

这里还有许多东西。哨位保证了所有前缀和后缀操作都能执行(21.3.8节)。我们通过调用`ostream`的成员函数`getloc()`(21.7节)取得它的现场,利用`use_facet`(D.3.1节)通过这个现场提取出`num_put`。在做了这些之后,我们就调用适当的`put()`函数去完成实际工作。从`ostream`可以创建相应的`ostreambuf_iterator`(19.2.6节),并可以隐式地将一个`ostream`转换到它的基类`ios_base`(21.2.1节),所以`put()`的前两个参数都很容易提供。

对`put()`的调用返回它的输出迭代器参数。这个输出迭代器是由`basic_ostream`得到的,所以它是一个`ostreambuf_iterator`。因此可以用`failed()`(19.2.6.1节)去检测失败,这也使我们能够去适当地设置流的状态。

我没有用`has_facet`,因为每个现场里都保证了所有标准刻画(D.4节)的存在。如果这种保证被违反,就会抛出`bad_cast`异常(D.3.1节)。

`put()`函数调用虚函数`do_put()`,因此就可能执行用户定义的代码,而`operator<<()`也必须准备去处理由覆盖的`do_put()`函数抛出的异常。还有,可能出现对某些字符类型不存在`num_put`的情况,所以`use_facet()`也可能抛出`std::bad_cast`异常(D.3.1节)。<<对内部类型(例如`double`)的行为方式由C++标准定义。因此,问题并不在于`handle_ioexception()`应该做什么,而是它应该怎样去做标准所规定的事情。如果这个`ostream`的异常状态里的`badbit`被置位(21.3.6节),该异常将被简单地再次抛出;否则,就处理这个异常,适当地设置流状态并继续下去。在两种情况中都要设置流状态中的`badbit`位(21.3.3节):

```
template<class Ch, class Tr>
void handle_ioexception(std::basic_ostream<Ch, Tr>& s) // 从catch子句中调用
{
    if (s.exceptions() & ios_base::badbit) {
        try { s.setstate(ios_base::badbit); } catch (...) { }
```

```

        throw;    // 重新抛出
    }
    s.setstate(ios_base::badbit);    // 可能抛出basic_ios::failure
}

```

这里需要用`try`块，因为`setstate()`可能抛出`basic_ios::failure`（21.3.3节、21.3.6节）。然而，如果在异常状态里的`badbit`被置位，`operator<<()`就必须重新抛出导致调用`handle_ioexception()`的那个异常（而不是简单地抛出`basic_ios::failure`）。

`double`一类的内部类型的`<<`必须通过直接写到流缓冲区的方式实现。在为用户定义类型写`<<`时，为避免结果的复杂性，我们可以基于已有类型的输出去描述用户定义类型的输出（D.3.2节）。

D.4.2.3 数值的输入

在从流缓冲区读的过程中（21.6.4节），`istream`依靠`num_get`刻画：

```

template <class Ch, class In = istreambuf_iterator<Ch> >
class std::num_get : public locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit num_get(size_t r = 0);

    // 将 [b:e) 读入 v，采用来自 s 的格式，通过设置 r 报告错误：
    In get(In b, In e, ios_base& s, ios_base::iostate& r, bool& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned short& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned int& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, unsigned long& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, float& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, ios_base& s, ios_base::iostate& r, void*& v) const;

    static locale::id id; // 刻画标识符对象（D.2节、D.3节、D.3.1节）

protected:
    ~num_get();

    // 为公用函数用的“do_”虚函数（见D.4.1节）
};

```

简而言之，`num_get`的组织方式与`num_put`类似（D.4.2.2节）。因为`get()`是读而不是写，因此它需要有一对输入迭代器，而且指明读入目标的参数是一个引用。这里将设置`iostate`变量`r`，以便反应流的状态。如果无法读入一个所需类型的值，就会设置`r`中的`failbit`；如果达到了输入结束，将设置到`r`中的`eofbit`。输入操作将利用`r`确定它的流的状态设置情况。如果未遇到错误，读到的值将通过`v`赋值，否则`v`将保持不变。

`istream`的实现者可能写出：

```

template <class Ch, class Tr>
istream& std::basic_istream<Ch, Tr>::operator>>(double& d)
{
    sentry guard(*this);    // 见21.3.8节
    if (!guard) {
        setstate(failbit);
        return *this;
    }
}

```

```

    }

    iostate state = 0;    // 好
    istreambuf_iterator<Ch> eos;
    double dd;
    try {
        use_facet< num_get<Ch> > (getloc()) . get (os, eos, *this, state, dd);
    }
    catch (...) {
        handle_ioexception(*this);    // 见D.4.2.2节
        return *this;
    }
    if (state==0 || state==eofbit) d = dd; // 当get() 成功时设置值
    setstate(state);
    return *this;
}

```

在出错的情况下 (21.3.6节), `setstate()` 将抛出那些为 `istream` 启用的异常。

通过重新定义 `num_punct`, 例如D.4.2节的 `My_punct`, 我们就能用非标准的标点读入了。例如,

```

void f()
{
    cout << "style A: "
    int i1;
    double d1;
    cin >> i1 >> d1;    // 使用标准的“12345678”格式读入

    locale loc(locale::classic(), new My_punct);
    cin.imbue(loc);
    cout << "style B: "
    int i2;
    double d2;
    cin >> i1 >> d2;    // 用“12 345 678”格式读入
}

```

如果想读入很不寻常的数值格式, 我们就必须覆盖 `do_get()` 函数。例如, 我们可以定义一个 `num_get`, 让它读入罗马数值, 例如 `XXI` 和 `MM` (D.5[15])。

D.4.3 货币值的输入和输出

从技术上看, 货币量的格式化问题与“普通”数值的格式化类似 (D.4.2节)。然而, 货币量的表示形式对文化差异却更加敏感。例如, 负值 (损失, 借方) 如 `-1.25`, 在某些环境中应该表示为括号里的 (正) 数 (`1.25`)。与此类似, 在有些环境中用颜色表示负值, 以便更容易识别。

没有标准的“货币类型”, 与此相反, 货币刻面就是为了程序员显式地对数值使用, 去表示已知为货币的数量。例如,

```

class Money { // 保存货币量的简单类型
    long int amount;
public:
    Money(long int i) : amount(i) { }
    operator long int() { return amount; }
};
// .

```

```
void f(long int i)
{
    cout << "value= " << i << " amount= " << Money(i) << endl;
}
```

货币刻面的工作就是使人们更容易为**Money**类型写输出运算符，使其数量能按照本地的习惯形式打印（见D.4.3.2节）。实际输出将依据**cout**的现场变化，可能的输出包括

```
value= 1234567 amount=$12345.67
value= 1234567 amount= 12345,67 DKK
value= -1234567 amount=$-12345.67
value= -1234567 amount= -$12345.67
value= -1234567 amount=(CHF12345,67)
```

对于货币，精确到最小货币单位通常被认为是至关重要的。因此，我采纳了一种常用的习惯，用整数值表示美分（便士、欧尔、费尔、分等）的数量，而不是美元（英镑、克朗、第纳尔、欧元等）的数量。这一习惯也得到**moneypunct**的**frac_digits()**函数的支持（D.4.3.1节）。与此类似，“小数点”的出现方式由**decimal_point()**定义。

剖面**money_get**和**money_put**提供了一些函数，它们能基于**money_base**剖面所定义的格式执行I/O操作。

简单的**Money**类型可以只用于控制I/O的格式，或者用于保存货币值。对于前一种情况，我们应该在写之前将用于保存货币值的（其他）类型强制到**Money**，而且先读入到**Money**变量，而后再转换到其他类型。始终如一地用**Money**类型保存货币值更不容易出错，按照这种方式，我们就不会忘记在写出之前强制到**Money**，也不会在试图以依赖于现场的方式读入货币值时出现输入错误。当然，将**Money**类型引进并非为处理货币而用的系统不一定可行，在这种情况下，采用读写操作中使用**Money**转换（强制）的方式就很有必要了。

D.4.3.1 货币的标点

自然，控制货币量表现形式的剖面**moneypunct**很像控制普通数值的剖面**num_punct**（D.4.2.1节）：

```
class std::money_base {
public:
    enum part { none, space, symbol, sign, value };    // 值布局的部分
    struct pattern { char field[4]; };                // 布局描述
};

template <class Ch, bool International = false>
class std::moneypunct : public locale::facet, public money_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit moneypunct(size_t r = 0);

    Ch decimal_point() const;                        // classic()里是','
    Ch thousands_sep() const;                        // classic()里是','
    string grouping() const;                          // classic()里是"", 表示不分组

    string_type curr_symbol() const;                 // classic()里是"$"
    string_type positive_sign() const;               // classic()里是""
    string_type negative_sign() const;               // classic()里是"-"

    int frac_digits() const;                          // 小数点后的数字个数, classic()里是2
    pattern pos_format() const;                       // classic()里是 {symbol, sign, none, value}
```

```

pattern neg_format() const;           // classic()里是 {symbol, sign, none, value}
static const bool intl = International; // 使用国际货币格式
static locale::id id; // 刻画标识符对象 (D.2节、D.3节、D.3.1节)

protected:
    ~moneypunct();
    // 为公用函数用的“do_”虚函数 (见D.4.1节)
};

```

由`moneypunct`提供的功能的主要意图是供`money_put`和`money_get`刻面的实现者使用 (D.4.3.2节、D.4.3.3节)。

`decimal_point()`、`thousands_sep()`和`group()`成员函数在`numput`里都有对应。

由`curr_symbol()`、`positive_sign()`和`negative_sign()`成员返回的字符串将分别被用于表示货币符号 (例如, \$、¥、FRF、DKK)、正号和负号。如果`International`模板参数是`true`, `intl`成员也将是`true`, 这时将使用货币符号的“国际”表示形式。这种“国际”表示形式都是4个字符的串, 例如

```

"USD "
"DKK"
"EUR "

```

最后一个字符是表示结束的0, 三字符的货币标识符由ISO 4217标准定义。当`International`为`false`时, 就使用“本地的”货币符号, 如\$、£和¥等。

由`pos_format()`或者`neg_format()`返回的`pattern`是包括四部分 (*part*) 的一个序列, 各部分分别表示数值、货币符号、正负号和空格的出现。最常用的格式都很容易借助于这种模式概念表示。例如,

```

+$123.45 // {sign, symbol, space, value} 其中positive_sign() 返回"+"
$+123.45 // {symbol, sign, value, none} 其中positive_sign() 返回"+"
$123.45 // {symbol, sign, value, none} 其中positive_sign() 返回""
$123.45- // {symbol, value, sign, none}
-123.45 DKK // {sign, value, space, symbol}
($123.45) // {sign, symbol, value, none} 其中negative_sign() 返回"("
(123.45DKK) // {sign, value, symbol, none} 其中negative_sign() 返回"("

```

通过让`negative_sign()`返回一个包含两个字符 () 的字符串的方式, 就可以得到用括号代表负数的形式。符号字符串的第一个字符将放到模式中`sign`所在的位置, 符号字符串的其余字符放在所有部分之后。这种功能最常见的用途就是表示金融领域用加括号形式表示负值的习惯, 其他使用也是可能的, 例如,

```

-$123.45 // {sign, symbol, value, none} 其中negative_sign() 返回"-"
*$123.45 silly // {sign, symbol, value, none} 其中negative_sign() 返回"* silly"

```

值`sign`、`value`和`symbol`必须在模式中各出现一次, 剩下的一个值可以是`space`或者`none`。如果出现`space`, 在表达形式中至少将出现一个空格字符, 也可能更多。如果出现的是`none`, 除了在模板最后之外, 表达形式中可以出现0个或者多个空格。

注意, 这一严格规则也禁止了一些明显合理的模式, 例如,

```

pattern pat = { sign, value, none, none }; // 错误: 没有symbol

```

函数`frac_digits()`指明小数点`decimal_point()`放在哪里。货币量通常采用最小的货币单位表示

(D.4.3节), 这个单位一般是货币中主要单位的1/100 (例如, 1¢ 是1\$ 的1/100), 所以 *frac_digits()* 常常是2。

下面是被定义为一个刻面的简单格式:

```
class My_money_io : public moneypunct<char, true> {
public:
    explicit My_money_io(size_t r = 0) : moneypunct<char, true>(r) {}

    Ch do_decimal_point() const { return "."; }
    Ch do_thousands_sep() const { return ","; }
    string do_grouping() const { return "\003\003\003"; }

    string_type do_curr_symbol() const { return "USD "; }
    string_type do_positive_sign() const { return ""; }
    string_type do_negative_sign() const { return " ("; }

    int do_frac_digits() const { return 2; } // 小数点之后2位数字

    pattern do_pos_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
    pattern do_neg_format() const
    {
        static pattern pat = { sign, symbol, value, none };
        return pat;
    }
};
```

这个侧面将在D.4.3.2节和D.4.3.3节所定义的*Money*输入和输出操作中使用。

还提供了*moneypunct*的一个*_byname*版本 (D.4节、D.4.1节)。

```
template <class Ch, bool Intl = false>
class std::moneypunct_byname : public moneypunct<Ch, Intl> { /* ... */ };
```

D.4.3.2 货币的输出

*money_put*侧面按照*moneypunct*描述的格式写出货币量。特别的, *money_put*提供了一些 *put()* 函数, 它们将适当格式化后的字符表示写进一个流的流缓冲区里:

```
template <class Ch, class Out = ostreambuf_iterator<Ch>>
class std::money_put : public std::locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_put(size_t r = 0);

    // 将值v放入由b定位的缓冲区;
    Out put(Out b, bool intl, ios_base& s, Ch fill, long double v) const;
    Out put(Out b, bool intl, ios_base& s, Ch fill, const string_type& v) const;

    static locale::id id; // 侧面标识符对象 (D.2节、D.3节、D.3.1节)

protected:
    ~money_put();

    // 为公用函数用的 "do_" 虚函数 (见D.4.1节)
};
```

这里***b***、***s***、***fill***和***v***参数的使用和***num_put***的***put()***完全一样（D.4.2.2节）。***intl***参数指明是使用标准的4字符“国际”货币表示，还是使用“本地”货币符号（D.4.3.1节）。

有了***money_put***，我们就可以定义***Money***（D.4.3节）的输出运算符了：

```
ostream& operator<<(ostream& s, Money m)
{
    ostream::sentry guard(s);          // 见21.3.8节
    if (!guard) return s;
    try {
        const money_put<char>& f = use_facet<money_put<char>>(s.getloc());
        if (m==static_cast<long double>(m)) { // m可以用long double表示
            if (f.put(s, true, s, s.fill(), m).failed()) s.setstate(ios_base::badbit);
        }
        else {
            ostringstream v;
            v << m;          // 转换到字符串表示
            if (f.put(s, true, s, s.fill(), v.str()).failed()) s.setstate(ios_base::badbit);
        }
    }
    catch (...) {
        handle_ioexception(s);          // 见D.4.2.2节
    }
    return s;
}
```

如果***long double***中没有足够的精度去准确表示货币值，我就将值转换到字符串表示，而后用以***string***为参数的***put()***输出。

D.4.3.3 货币的输入

money_get刻画按照***money_punct***描述的格式读入货币量。特别地，***money_get***提供了一些***get()***函数，它们将从一个流的流缓冲区里提取出具有适当格式的字符表示：

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class std::money_get : public std::locale::facet {
public:
    typedef Ch char_type;
    typedef In iter_type;
    typedef basic_string<Ch> string_type;

    explicit money_get(size_t r = 0);

    // 将 [b:e) 读入v，采用来自s的格式规则，通过设置r报告错误：
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, long double& v) const;
    In get(In b, In e, bool intl, ios_base& s, ios_base::iostate& r, string_type& v) const;

    static locale::id id; // 刻画标识符对象（D.2节、D.3节、D.3.1节）
protected:
    ~money_get();

    // 为公用函数用的“do_”虚函数（见D.4.1节）
};
```

这里***b***、***e***、***s***、***fill***和***v***参数的使用和***num_get***的***get()***完全一样（D.4.2.3节）。***intl***参数指明是使用标准的4字符“国际”货币表示，还是使用“本地”货币符号（D.4.3.1节）。

定义良好的一对***money_get***和***money_put***刻画将提供一种输出形式，使它可以重新读回而不会出错或丢失信息。例如，


```
int main()
{
    Money m;
    while (cin>>m) cout << m << "\n";
}
```

这个简单程序的输出应该能接受作为它的输入。进一步说，将第一次运行的输出给第二次运行作为输入，产生的输出应该与第一次的一样。

*Money*的一个可能的输入运算符是：

```
istream& operator>>(istream& s, Money& m)
{
    istream::sentry guard(s);    // 见21.3.8节
    if (!guard) {
        s.setstate(ios_base::failbit);
        return s;
    }

    ios_base::iostate state = 0;    // 好
    istreambuf_iterator<char> eos;
    double dd;
    try {
        const money_get<char> &f = use_facet<money_get<char>>(s.getloc());
        f.get(s, eos, true, state, dd);
    }
    catch (...) {
        handle_ioexception(s);    // 见D.4.2.2节
        return s;
    }
    if (state==0 || state==ios_base::eofbit) m = dd; // 当get()成功时设置值
    s.setstate(state);
    return s;
}
```

D.4.4 日期和时间的输入输出

不幸的是，C++标准库并没有提供一个完美的*date*类型。但无论如何，它从C标准库继承来一些处理日期和时间间隔的低级功能，这些C功能可以成为C++按照与系统无关的方式处理时间的基础。

下面几节将展示如何使日期和时间信息的表现形式依赖于所用的*locale*。除此之外，它们还提供了一个示例，说明了怎样将一个用户定义类型（*Date*）融入由*iostream*（第21章）和*locale*（D.2节）构筑的框架之中。*Date*的实现也展示了一些有用的技术，可供你在没有可用的*Date*类型的情况下处理时间时使用。

D.4.4.1 时钟和计时器

在最低的层次上，大部分系统都有一个极细粒度的计时器。标准库提供了函数*clock()*，它返回某个由实现定义的算术类型*clock_t*。*clock()*的结果可以用宏*CLOCKS_PER_SEC*校准。如果你无法访问某个可靠的计时功能，那么就可以用下面方式实测一个循环：

```
int main(int argc, char* argv[])    // 见6.1.7节
{
    int n = atoi(argv[1]);          // 见20.4.1节
    clock_t t1 = clock();
```

```

    if (t1 == clock_t(-1)) {          // clock(-1) 表示 "clock() 不工作"
        cerr << "sorry, no clock\n";
        exit(1);
    }

    for (int i = 0; i < n; i++) do_something(); // 为循环计时

    clock_t t2 = clock();
    if (t2 == clock_t(-1)) {
        cerr << "sorry, clock overflow\n";
        exit(2);
    }
    cout << "do_something() " << n << " times took "
         << double(t2 - t1) / CLOCKS_PER_SEC << " seconds"
         << " (measurement granularity: " << CLOCKS_PER_SEC << " of a second)\n";
}

```

在除法之前的显式转换`double(t2 - t1)`是必须的, 因为`clock_t`可能是个整数类型。`clock()`到底从什么时间开始运行由实现定义; `clock()`的设计意图是在程序的一次运行中实测时间的间隔。对于由`clock()`返回的值`t1`和`t2`, `double(t2 - t1)/CLOCKS_PER_SEC`是系统对两次调用之间按秒计时所得时间的最好近似。

如果某处理器未提供`clock()`, 或者被测时间间隔过长, `clock()`就会返回`clock(-1)`。

`clock()`函数的设计意图是用于实测时间间隔, 从不到一秒直至若干秒。举个例子, 假设`clock_t`是32位的`int`, `CLOCKS_PER_SEC`是1 000 000, 我们用`clock()`将只能以微秒为单位从0计起直到稍大于2000秒(大约为半小时)。

请注意, 取得对程序的有意义的实测值是一种相当技巧性的问题。在同一机器上运行的其他程序可能严重地影响一次运行所耗费的时间, 高速缓存和流水线的影响难以预计, 算法也可能对数据的情况有很大的依赖性。如果你试图要对什么计时, 请多运行几次, 并应拒绝那些运行时间变化很显著的结果。

为了对付较长的时间间隔和日历时间, 标准库提供了一个类型`time_t`, 用以表示时间点; 还提供了结构`tm`, 将一个时间点的信息划分为一些习惯的部分:

```

typedef implementation_defined time_t;    // 实现定义的算术类型 (4.1.1 节)
                                           // 能表示一段时间
                                           // 通常为32位整数

struct tm {
    int tm_sec;      // 分内的秒数 [0, 61]; 60和61表示闰秒
    int tm_min;      // 时内的分钟数 [0, 59]
    int tm_hour;      // 一日内的时序数 [0, 23]
    int tm_mday;      // 一月内的日序数 [1, 31]
    int tm_mon;       // 年内的月份 [0, 11]; 0意味着一月 (注意: 不是 [1, 12])
    int tm_year;      // 从1900开始的年份; 0表示1900, 102表示2002
    int tm_wday;      // 从星期天开始的日序数 [0, 6]; 0表示星期天
    int tm_yday;      // 从一月一日开始的日序数 [0, 365]; 0表示1月1日
    int tm_isdst;     // 夏季时的小时数
};

```

注意, 标准只保证`tm`里存在这里所提到的所有`int`成员, 但它并不保证这些成员按上述顺序出现, 也不保证没有其他成员。

`time_t`和`tm`类型以及使用它们的基本功能都在`<ctime>`和`<time.h>`里给出。例如,

```

clock_t clock();          // 程序开始运行起的嘀嗒次数

```

```

time_t time(time_t* pt);           // 当前日历时间
double difftime(time_t t2, time_t t1); // t2 - t1的秒数

tm* localtime(const time_t* pt);    // *pt的本地时间
tm* gmtime(const time_t* pt);       // *pt的格林威治时间 (GMT) tm, 或者0
// (官方称世界协调时间, UTC)

time_t mktime(tm* ptm);            // *ptm的time_t或者time_t(-1)

char* asctime(const tm* ptm);       // *ptm的C风格字符串表示
// 如 "Sun Sep 16 01:03:52 1973\n"

char* ctime(const time_t* t) { return asctime(localtime(t)); }

```

当心：`localtime()` 和 `gmtime()` 都返回一个 `tm*`，它们实际指向一个静态分配的对象；随后再去调用函数就会改变这个对象的值。请立即使用这个返回值或者是将这个 `tm` 复制到你控制的存储区里。与此类似，`asctime()` 返回一个指向静态分配的字符数组的指针。

一个 `tm` 可以表示至少数万年范围内的日期（对最小的 `int` 是大约 $[-32000, 32000]$ ）。然而，最常见的 `time_t` 是（有符号的）32位 `long int`。按照秒计算，这将使 `time_t` 能够在基础年份的前后各计算刚刚超过68年的时间。最常见的基础年份是1970，所用的准确基础时间是一月一日的0:00 (GMT)。如果 `time_t` 是32位有符号的整数，我们将在2038年用完我们的时间，除非是更新到更大的整数，正如许多系统上已经做了的那样。

`time_t` 机制基本上是想用于表示“最近的当前时间”。这样，我们就不能希望用 `time_t` 去表示超出 $[1902, 2038]$ 范围的时间。更糟糕的是，并不是所有函数的实现都能按照同样的方式处理负值。为了可移植性，需要同时表示为 `tm` 和 `time_t` 的值就应该位于 $[1970, 2038]$ 的范围中。想表示超出1970到2038这段时间的人，就必须自己设计某种方式去做有关的事情。

这种情况的一个结论就是 `mktime()` 可能失败。如果 `mktime()` 的参数值无法表示为 `time_t`，返回的就是指明失败的 `time_t(-1)`。

如果我们有一个运行时间很长的程序，那么就可以采用如下方式为其计时：

```

int main(int argc, char* argv[]) // 见6.1.7节
{
    time_t t1 = time(0);
    do_a_lot(argc, argv);
    time_t t2 = time(0);
    double d = difftime(t2, t1);
    cout << "do_a_lot() took" << d << " seconds\n";
}

```

如果 `time()` 的参数不是0，结果时间也将赋值给参数指针所指的那个 `time_t` 变量。如果日历时间不能使用（譬如说，在某个特殊处理器上），返回的值是 `time_t(-1)`。我们可以试用如下方式，在确定今天的日期时加一点小心：

```

int main()
{
    time_t t;

    if (time(&t) == time_t(-1)) { // time_t(-1) 说明 "time() 不能用"
        cerr << "Bad time\n";
        exit(1);
    }

    tm* gt = gmtime(&t);
    cout << gt->tm_mon+1 << '/' << gt->tm_mday << '/' << 1900+gt->tm_year << endl;
}

```

D.4.4.2 一个Date类

如10.3节所述, 一个Date类型不可能服务于所有的目的。日期信息的使用将会要求各种各样的表示形式, 在19世纪以前的日历信息就与历史的反复无常密切相关。然而, 作为一个示例, 我们还是可以沿着10.3节的思路定义一个Date类型, 采用time_t作为其实现:

```
class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
    class Bad_date {};
    Date(int dd, Month mm, int yy);
    Date();

    friend ostream& operator<<(ostream& s, const Date& d);
    // ...
private:
    time_t d; // 标准日期与时间表示
};

Date::Date(int dd, Month mm, int yy)
{
    tm x = { 0 };
    if (dd<0 || 31<dd) throw Bad_date(); // 过分简化: 见10.3.1节
    x.tm_mday = dd;
    if (mm<jan || dec<mm) throw Bad_date();
    x.tm_mon = mm-1; // tm_mon从0开始
    x.tm_year = yy-1900; // tm_year从1900开始
    d = mktime(&x);
}

Date::Date()
{
    d = time(0); // 默认日期: 今天
    if (d == time_t(-1)) throw Bad_date();
}
```

下面的工作是为Date实现对现场敏感的输入输出 << 和 >>。

D.4.4.3 日期和时间输出

与num_put(D.4.2节)一样, time_put提供了一些put()函数, 它们通过迭代器向缓冲区里写信息:

```
template <class Ch, class Out = ostreambuf_iterator<Ch> >
class std::time_put : public locale::facet {
public:
    typedef Ch char_type;
    typedef Out iter_type;

    explicit time_put(size_t r = 0);

    // 将通过b写入s的流缓冲区, 采用格式fmt;
    Out put(Out b, ios_base& s, Ch fill, const tm* t,
            const Ch* fmt_b, const Ch* fmt_e) const;

    Out put(Out b, ios_base& s, Ch fill, const tm* t, char fmt, char mod = 0) const
        { return do_put(b, s, fill, t, fmt, mod); }

    static locale::id id; // 刻面标识符对象 (D.2节、D.3节、D.3.1节)
```

```
protected:
    ~time_put();

    virtual Out do_put(Out, ios_base&, Ch, const tm*, char, char) const;
};
```

调用`put(b, s, fill, fmt_b, fmt_e)` 将把取自`t`的日期信息通过`b`放入`s`的流缓冲区里。`fill`字符用在所有需要填充的地方。输出格式由类似`printf()`的格式串`[fmt_b, fmt_e)`描述。这种类似`printf` (21.8节)的格式用于产生实际输出, 其中可包括下面这些专用格式描述符:

- `%a` 缩写的星期名 (例如, Sat)。
- `%A` 完整星期名 (例如, Saturday)。
- `%b` 缩写月份名 (例如, Feb)。
- `%B` 完整月份名 (例如, February)。
- `%c` 日期和时间 (例如, Sat Feb 06 21:46:05 1999)。
- `%d` 月中的日序数 [01, 31] (例如, 06)。
- `%H` 按24小时计的钟点 [00, 23] (例如, 21)。
- `%I` 按12小时计的钟点 [00, 11] (例如, 09)。
- `%j` 年中的日序数 [001, 366] (例如, 037)。
- `%m` 年中的月份 [01, 12] (例如, 02)。
- `%M` 小时中的分数 [00, 59] (例如, 48)。
- `%p` 12小时计时方式的a.m./p.m.指示 (例如, PM)。
- `%S` 分中的秒数 [00, 61] (例如, 40)。
- `%U` 一年中从周日开始的星期数 [00, 53] (例如, 05), 从第一个周日开始第一周。
- `%w` 一周里的日序数 [0, 6]; 0表示星期天 (例如, 6)。
- `%W` 一年中从周一开始的星期数 [00, 53] (例如, 05), 从第一个周一开始第一周。
- `%x` 日期 (例如, 02/06/99)。
- `%X` 时间 (例如, 21:48:40)。
- `%y` 不包含世纪的年份 [00, 99] (例如, 99)。
- `%Y` 年份 (例如, 1999)。
- `%Z` 时区指示符 (例如, EST), 如果知道时区。

这个长长的专用格式规则列表可以用做参数, 为扩展的I/O系统所用。当然, 就像大部分专用的记法形式一样, 它对于自己的工作最合适, 常常也最方便。

除了这些格式化指示符之外, 许多系统还支持一些“修饰符”, 例如, 用整数描述的域宽度 (21.8节), 如`%10X`等。对于时间和日期的修饰符格式并不是C++标准的一部分, 但也有些平台标准 (如POSIX) 要求它们。因此, 修饰符的使用将很难避免, 即使它们不是完全可移植的。

来自`<ctime>`和`<time.h>`的函数`strftime()`与`sprintf()` (21.8节)类似, 它采用时间和日期指示符产生输出:

```
size_t strftime(char* s, size_t max, const char* format, const tm* tmp);
```

这一函数将根据`format`, 将由`*tmp`和`format`产生的至多`max`个字符放入`*s`。例如,

```
int main()
{
```

```

char buf[20]; // 懒散; 但愿strftime()将绝不会产生超过20个字符
time_t t = time(0);
strftime(buf, 20, "%A\n", localtime(&t));
cout << buf;
}

```

在某个星期三, 如果使用默认的*classic()*现场(D.2.3节), 这将打印出*Wednesday*; 如果用丹麦现场, 就会打印出*onsdag*。

所有非格式描述符的字符(例如上例中的换行符)都将被直接复制到第一个参数(*s*)。

当*put()*识别出一个格式字符*f*时(可能还有可选的修饰符字符*m*), 它就调用虚函数*do_put()*去做实际格式化输出: *do_put(b, s, fill, f, m)*。

调用*put(b, s, fill, t, f, m)*是*put()*的简化形式, 其中显式地提供了格式字符(*f*)和修饰符字符(*m*)。这样,

```

const char fmt[] = "%10X";
put(b, s, fill, t, fmt, fmt+sizeof(fmt));

```

就可以简化为

```

put(b, s, fill, t, 'X', 10);

```

如果一个格式中包含了多字节字符, 它必须在默认状态中开始和结束(D.4.6节)。

我们可以用*put()*为*Date*实现一个对*locale*敏感的输出运算符了:

```

ostream& operator<<(ostream& s, const Date& d)
{
    ostream::sentry guard(s); // 见21.3.8节
    if (!guard) return s;

    tm* tmp = localtime(&d.d);
    try {
        if (use_facet<time_put<char>>(s.getloc()).put(s, s, s.fill(), tmp, 'x').failed())
            s.setstate(ios_base::failbit);
    }
    catch (...) {
        handle_ioexception(s); // 见D.4.2.2节
    }
    return s;
}

```

因为不存在标准的*Date*类型, 所以也就不存在日期I/O的默认布局方式。在这里, 我描述*%x*格式的方式是将字符'*x*'作为格式字符传递。因为*%x*是*get_time()*的默认格式(D.4.4.4节), 这样做可能已经是最大限度地接近标准了。参见D.4.4.5节中使用其他格式的例子。

同时提供了*time_put*的一个*_byname*版本(D.4节、D.4.1节):

```

template<class Ch, class Out = ostreambuf_iterator<Ch>>
class std::time_put_byname : public time_put<Ch, out> { /* ... */ }

```

D.4.4.4 日期和时间输入

像平常一样, 输入比输出更具有技巧性。因为在我们写代码输出一个值时, 我们还常常可以在几种可能格式中做选择。进一步说, 我们在写输入代码时还必须处理错误, 有时必须处理出现多种格式的可能性。

*time_get*刻画实现时间和日期的输入。其思想是: 一个*locale*的*time_get*可以读入由这个*locale*的*time_put*产生的时间和日期。当然, 并不存在标准的*date*和*time*类, 因此程序员可以

用一个`locale`，去按照可能变化的格式产生输出。举个例子，下面的所有表示形式可能都是由同一个输出语句产生的，其中用的是不同现场的`time_put`（D.4.4.5节）：

```
January 15th 1999
Thursday 15th January 1999
15 Jan 1999AD
Thurs 15/1/99
```

C++标准鼓励`time_get`的实现者去接受由POSIX或者其他标准所描述的日期和时间格式。问题是，按照给定文化的习惯格式去读日期和时间的意图很难进行标准化。明智的做法是先做些试验，看看现场给出什么东西（D.6[8]）。如果某个格式不能接受，程序员就可以在提供一个可供选择的适当的`time_get`剖面。

标准的时间输入`facet`是`time_get`，它是从`time_base`派生而来：

```
class std::time_base {
public:
    enum dateorder {
        no_order, // 无顺序，可能有更多元素（如一周中的日数）
        dmy,      // 按日月年排列
        mdy,      // 按月日年排列
        ymd,      // 按年月日排列
        ydm       // 按年日月排列
    };
};
```

实现可以借助于这个枚举去简化对日期格式的分析。

与`num_get`一样，`time_get`也通过一对输入迭代器访问其缓冲区：

```
template <class Ch, class In = istreambuf_iterator<Ch> >
class time_get : public locale::facet, public time_base {
public:
    typedef Ch char_type;
    typedef In iter_type;

    explicit time_get(size_t r = 0);

    dateorder date_order() const { return do_date_order(); }

    // 将(b:e)读入d，采用来自s的格式，通过设置r报告错误：
    In get_time(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_year(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    In get_weekday(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;
    In get_monthname(In b, In e, ios_base& s, ios_base::iostate& r, tm* d) const;

    static locale::id id; // 剖面标识符对象（D.2节、D.3节、D.3.1节）

protected:
    ~time_get();

    // 为公用函数用的“do_”虚函数（见D.4.1节）
};
```

`get_time()` 函数调用`do_get_time()`。默认的`get_time()`采用`%X`格式，按照同一`locale`的`time_put::put()`产生输出的格式去读入时间（D.4.4节）。类似的，`get_date()`函数调用的是`do_get_date()`。默认的`get_date()`是采用`%x`格式去读入一个日期，按同一`locale`的`time_put::put()`产生输出所用格式（D.4.4节）去读。

这样，为*Date*写的最简单的输入运算符可能就是下面样子：

```
istream& operator>>(istream& s, Date& d)
{
    istream::sentry guard(s);    // 见21.3.8节
    if (!guard) return s;

    ios_base::iostate res = 0;
    tm x = { 0 };
    istreambuf_iterator<char, char_traits<char>> > end;
    try {
        use_facet<time_get<char>>(s.getloc()).get_date(s, end, s, res, &x);
    }
    catch (...) {
        handle_ioexception(s);    // 见D.4.2.2节
        return s;
    }
    d = Date(x.tm_mday, Date::Month(x.tm_mon+1), x.tm_year+1900);
    return s;
}
```

调用*get_date(s, end, s, res, &x)*，要依靠来自*istream*的两个隐式转换：作为第一个参数的*s*被用于创建一个*istreambuf_iterator*，作为第三个参数的*s*被转换到*istream*的基类*ios_base*。

在能由*time_t*表示的日期范围内，这个输入运算符都能正确工作。

下面是一个简单的测试程序：

```
int main()
try {
    Date today;
    cout << today << endl;    // 用%x格式写出
    Date d(12, Date::may, 1998);

    cout << d << endl;
    Date dd;
    while (cin >> dd) cout << dd << endl;    // 读入用%x格式产生的日期
}
catch (Date::Bad_date) {
    cout << "exit: bad date caught\n";
}
```

同时提供了*time_get*的一个*_byname*版本（D.4节、D.4.1节）：

```
template <class Ch, class Out = ostreambuf_iterator<Ch>>
class std::time_put_byname : public time_put<Ch, Out> { /* ... */ };
```

D.4.4.5 一个更灵活的*Date*类

如果你试着用来自D.4.4.2节的*Date*类和D.4.4.3节与D.4.4.4节的I/O，你很快就会发现它的局限性：

- [1] 它只能处理*time_t*能表示的日期，典型情况的范围是在 [1970, 2038] 内。
- [2] 它只接受标准格式的日期——无论那是什么。
- [3] 它对输入错误的报告方式无法让人接受。
- [4] 它只支持*char*的流——不是任意字符类型的流。

一个更有趣也更有用的输入运算符应该能接受更广范围的日期，识别出一些常见的格式，并以某种有用的方式可靠地报告错误。为此，我们就必须离开*time_t*表示：


```

class Date {
public:
    enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

    struct Bad_date {
        const char* why;
        Bad_date(const char* p) : why(p) { }
    };

    Date(int dd, Month mm, int yy, int day_of_week = 0);
    Date();

    void make_tm(tm* t) const;    // 将Date的tm表示放在 *t
    time_t make_time_t() const;  // 返回Date的time_t表示

    int year() const { return y; }
    Month month() const { return m; }
    int day() const { return d; }

    // ...
private:
    char d;
    Month m;
    int y;
};

```

为简单起见，我转到 (d, m, y) 表示（10.2节）。

构造函数可以如下定义：

```

Date::Date(int dd, Month mm, int yy, int day_of_week)
    : d(dd), m(mm), y(yy)
{
    if (d==0 && m==Month(0) && y==0) return;    // Date(0,0,0) 是“空日期”
    if (mm<jan || dec<mm) throw Bad_date("bad month");
    if (dd<1 || 31<dd) // 过分简化，见10.3.1节
        throw Bad_date("bad day of month");
    if (day_of_week && day_in_week(yy, mm, dd) != day_of_week)
        throw Bad_date("bad day of week");
}

Date::Date() : d(0), m(0), y(0) { } // 一个“空日期”

```

`day_in_week()` 计算并不简单，但对`locale`机制而言则无关大局，所以我将它放到一边。如果你需要它，一定可以在系统里的某个地方找到它。

对于像`Date`这样的类型，比较操作总是很有用的：

```

bool operator==(const Date& x, const Date& y)
{
    return x.year()==y.year() && x.month()==y.month() && x.day()==y.day();
}

bool operator!=(const Date& x, const Date& y)
{
    return !(x==y);
}

```

由于脱离了标准的`tm`和`time_t`格式，我们需要有相应转换函数，以便与要求使用这些类型的软件相互合作：

```

void Date::make_tm(tm* p) const    // 将日期放入*p
{
    tm x = { 0 };
    *p = x;
    p->tm_year = y-1900;
    p->tm_mday = d;
    p->tm_mon = m-1;
}

time_t Date::make_time_t() const
{
    if (y<1970 || 2038<y)        // 过分简化
        throw Bad_date("date out of range for time_t");
    tm x;
    make_tm(&x);
    return mktime(&x);
}

```

D.4.4.6 描述一种Date格式

C++没有为日期定义一种标准的输出格式（%x是我们能得到的最接近标准的东西；D.4.4.3节）。当然，即使存在某种标准格式，我们可能也会想采用其他格式。要做到这一点，可以通过提供一个“默认格式”和一个修改它的方式。例如，

```

class Date_format {
    static char fmt[];           // 默认格式
    const char* curr;           // 当前格式
    const char* curr_end;
public:
    Date_format() : curr(fmt), curr_end(fmt+strlen(fmt)) {}

    const char* begin() const { return curr; }
    const char* end() const { return curr_end; }

    void set(const char* p, const char* q) { curr=p; curr_end=q; }
    void set(const char* p) { curr=p; curr_end=curr+strlen(p); }

    static const char* default_fmt() { return fmt; }
};

const char Date_format::fmt[] = "%A, %B %d, %Y"; // 例如，Friday, February 5, 1999

Date_format date_fmt;

```

为了使用`strftime()`格式（D.4.4.3节），我抑止住自己，没有将`Date_format`所用的字符类型参数化。这也意味着这个解决方案只允许能表示为`char[]`的数据格式。在这里，我还用一个全局格式对象（`date_fmt`）提供了一种默认的`Date`格式。因为`date_fmt`的值可以改变，这就提供了一种很生硬的控制`Date`格式的方式，类似于通过`global()`（D.2.3节）控制格式的方式。

一种更具一般性的解决方案是增添`Date_in`和`Date_out`刻画，分别去控制对流的读和写。这种途径将在D.4.4.7节介绍。

有了`Date_format`，`Date::operator<<()`可以写成下面的样子：

```

template<class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<<(basic_ostream<Ch, Tr>& s, const Date& d)
// 依照用户确定的格式写
{
    typename basic_ostream<Ch, Tr>::sentry guard(s); // 参见_io.sentry
    if (!guard) return s;
}

```

```

tm t;
d.make_tm(&t);
try {
    const time_put<Ch>&f = use_facet<time_put<Ch>>(s.getloc());
    if (f.put(s, s, s.fill(), &t, date_fmt.begin(), date_fmt.end()).failed())
        s.setstate(ios_base::failbit);
}
catch (...) {
    handle_ioexception(s);    // 见D.4.2.2节
}
return s;
}

```

我也可以利用`has_facet`去检验`s`的现场里确实有`time_put<Ch>`剖面；然而在这里用捕捉由`use_facet`抛出的任何异常的方式处理问题，看起来更简单一些。

下面是一个简单的测试程序，其中通过`date_fmt`控制输出格式：

```

int main()
try {

    while (cin >> dd && dd != Date()) cout << dd << endl;    // 按默认date_fmt写
    date_fmt.set("%Y/%m/%d");
    while (cin >> dd && dd != Date()) cout << dd << endl;    // 按"%Y/%m/%d"格式写
}
catch (Date::Bad_date e) {
    cout << "bad date caught: " << e.why << endl;
}

```

D.4.4.7 一个Date输入剖面

与通常的情况一样，输入总是比输出更困难一点。然而，因为与低层输入的界面已由`get_date()`固定下来，也因为在D.4.4.4节中为`Date`定义的`operator>>()`并没有直接访问`Date`的表示，我们就可能继续使用不加改动的`operator>>()`。这里是一个与`operator<<()`相匹配的模板版本：

```

template<class Ch, class Tr>
istream<Ch, Tr>& operator>>(istream<Ch, Tr>& s, Date& d)
{
    typename istream<Ch, Tr>::sentry guard(s);
    if (!guard) return s;

    ios_base::iostate res = 0;
    tm x = { 0 };
    istreambuf_iterator<Ch, Tr> end;
    try {
        use_facet<time_get<Ch>>(s.getloc()).get_date(s, end, s, res, &x);
    }
    catch (...) {
        handle_ioexception(s);    // 见D.4.2.2节
        return s;
    }
    d = Date(x.tm_mday, Date::Month(x.tm_mon+1), x.tm_year+1900, x.tm_wday);
    if (res == ios_base::badbit) s.setstate(res);
    return s;
}

```

这一Date输入运算符调用了istream的time_get 刻面的get_date(), 因此, 我们就可以通过定义由time_get派生出的新刻面的方式提供不同的更灵活的输入形式:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
class Date_in : public std::time_get<Ch, In> {
public:
    Date_in(size_t r = 0) : std::time_get<Ch, In>(r) {}

protected:
    In do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const;

private:
    enum Vtype { novalue, unknown, dayofweek, month };
    In getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const;
};
```

函数getval() 要求读入一个年份、一个月份、该月中一个日期, 以及可选的该日期的星期几, 并将这些结果组合到一个tm里。

月份名和一周中星期几的名字都由现场确定。因此我们就不必在输入函数里直接提到它们了。与此相反, 我们将通过调用由time_get为此提供的函数get_monthname() 和get_weekday() (D.4.4.4节) 去识别月份和星期几。

年份、月份中的日期, 或许还有月份都表示为整数。不幸的是, 一个数本身无法指出自己表示的是日期、月份还是别的什么。例如, 7可能表示七月、一个月的第七天, 甚至可能是2007年。time_get中date_order() 的实际用途就是消解这种歧义性。

Date_in的策略是读入一些值, 对它们进行分类, 而后用date_order() 去检查所得到的这些值是否 (以及怎样) 具有意义。私用的getval() 函数完成从流缓冲区实际读入以及初始分类的工作:

```
template<class Ch, class In = istreambuf_iterator<Ch> >
In Date_in::getval(In b, In e, ios_base& s, ios_base::iostate& r, int* v, Vtype* res) const
// 读Date的各部分: number、date_of_week或month。跳过空白
{
    ctype<Ch> const& ct = use_facet<ctype<Ch>>(s.getloc());
    Ch c = *b;
    if (r.skipws) {
        while (ct.isspace(c) || ct.ispunct(c)) { // 跳过空格和标点
            if (++b == e) {
                *res = novalue; // 没找到值
                return e;
            }
            c = *b;
        }
    }
    if (ct.isdigit(c)) { // 读入整数, 不管标点
        int i = 0;

        do { // 将来自任意字符集的数字转换到十进制值:
            static char const digits[] = "0123456789";
            i = i*10 + find(digits, digits+10, ct.narrow(c, ' ')) - digits;
            c = *++b;
        } while (ct.isdigit(c));
```

```

        *v = i;
        *res = unknown;    // 一个整数，但我们不知道它表示什么
        return b;
    }

    if (ct.isalpha(c)) {    // 找月份名或者星期几
        basic_string<Ch> str;
        while (ct.isalpha(c)) {    // 向字符串中读入字符
            str += c;
            if (++b == e) break;
            c = *b;
        }

        tm t;
        basic_stringstream<Ch> ss(str);
        get_monthname(ss.rdbuf(), In(), s, r, &t); // 从在内存的流缓冲区读
        if ((r & (ios_base::badbit | ios_base::failbit)) == 0) {
            *v = t.tm_mon;
            *res = month;
            r = 0;
            return b;
        }

        r = 0;    // 在试图读下一时间前清状态
        get_weekday(ss.rdbuf(), In(), s, r, &t);    // 从在内存的流缓冲区读
        if ((r & ios_base::badbit) == 0) {
            *v = t.tm_wday;
            *res = dayofweek;
            r = 0;
            return b;
        }
    }
    r |= ios_base::failbit;
    return b;
}

```

这里的技巧部分是将月份与星期几区分开。我们是通过输入迭代器读，所以无法去读 $[b, e)$ 两次，无法先读月份而后再看日。另一方面，我们也不能一次读一个字符并做判断，因为只有 `get_monthname()` 和 `get_weekday()` 知道在给定的现场里，哪些字符序列构成了月份的名字和一星期里各天的名字。在这个解中我选择的是先将字母字符的串读入一个 `string`，再由此串做出一个 `stringstream`，而后再反复从这个流的 `streambuf` 读出。

错误记录仍直接使用状态位，例如 `ios_base::badbit`。这是必须的，因为那些使用方便的对流状态执行操作的函数（如 `clear()` 和 `setstate()` 等）都是定义在 `basic_ios` 里，而不是定义在它的基类 `ios_base` 里（21.3.3节）。如果需要，这个 `>>` 运算符将使用 `get_date()` 报告的错误结果，并重新设置输入流的状态。

有了 `getval()`，我们就可以首先读入这些值，而后试着去看它们具有怎样的意义。`date_order()` 可能很关键：

```

template<class Ch, class In
In Date_in::do_get_date(In b, In e, ios_base& s, ios_base::iostate& r, tm* tmp) const
// 可选的星期几，后面是ymd, dmy, mdy或者ydm
{
    int val[3];    // 为以某种顺序给出日、月、年
    Vtype res[3] = { novalue };    // 用于值的分类

```

```

for (int i=0; b!=e && i<3; ++i) { // 读入日、月、年
    b = getval(b, e, s, r, &val[i], &res[i]);
    if (r) return b; // 呜呼：错误
    if (res[i]==novalue) { // 不能完成一个日期
        r |= ios_base::badbit;
        return b;
    }
    if (res[i]==dayofweek) {
        tmp->tm_wday = val[i];
        --i; // 呜呼：不是一个日、月或者年
    }
}

time_base::dateorder order = date_order(); // 现在试着弄清所读到的值的意义
if (res[0] == month) { // mdy或者错误
    // ...
}
else if (res[1] == month) { // dmy或ymd或错误
    tmp->tm_mon = val[1];
    switch (order) {
        case dmy:
            tmp->tm_mday = val[0];
            tmp->tm_year = val[2];
            break;
        case ymd:
            tmp->tm_year = val[0];
            tmp->tm_mday = val[2];
            break;
        default:
            r |= ios_base::badbit;
            return b;
    }
}
else if (res[2] == month) { // ydm或错误
    // ...
}
else { // 依靠dateorder或是错误
    // ...
}

tmp->tm_year -= 1900; // 调整基础年份以适合tm的规定
return b;
}

```

我忽略了一些代码段，它们对于理解现场、日期以及对输入的处理没有什么帮助。写出更好更一般的输入函数的问题留做练习（D.6[9-10]）。

下面是一个简单的测试程序：

```

int main()
try {
    cin.imbue(loc(locale()), new Date_in()); // 用Date_in读入Date
    while (cin >> dd && dd != Date()) cout << dd << endl;
}
catch (Date::Bad_date e) {
    cout << "bad date caught: " << e.why << endl;
}

```

注意, `do_get_date()` 将会接受一些无意义的日期, 例如,

Thursday October 7, 1998

和

1999/Feb/31

对年、月、日及可选的星期几的一致性检查将由 *Date* 的构造函数完成。*Date* 类的工作是构造出正确的日期, 而 *Date_in* 没有必要去共享这方面的知识。

也可能让 `getval()` 或者 `do_get_date()` 去猜测数值的意义。例如,

12 May 1922

很清楚, 不会有12年的1922日。也就是说, 我们能“猜出”这个日子不会是某个特定月的一天, 必然是年份。这种“猜测”在特殊的受限的环境中也可能有用。然而, 在更广泛的环境中就未必是好主意了。例如,

12 May 15

可能是年份12, 15, 1912, 1915, 2012或2015中的一个日子。有时更好的方式是扩充记法, 增加有关年份或者日子的线索。例如, *1st*或者*15th*很清楚是某月中的第几天。与此类似, *751BC*和*1453AD*显然表示年份。

D.4.5 字符类别

在从输入读字符时, 经常需要对它们进行分类, 以便确定读到的内容是否有意义。例如, 在读入一个数时, 输入例程就需要知道哪些字符是数字。类似地, 6.1.2节显示了如何用标准的字符分类函数去分析输入。

很自然, 字符的类别依赖于所用的字母表。因此, 现场里提供了 *ctype* 剖面, 以便表示各个字符的类别。

字符类通过一个名为 *mask* 的枚举描述:

```
class std::ctype_base {
public:
    enum mask {                // 实际值由实现定义
        space = 1,             // 空白 (在 "C" 现场里: ' ', '\n', '\t', ...)
        print = 1<<1,          // 可打印字符
        cntrl = 1<<2,           // 控制字符
        upper = 1<<3,           // 大写字母
        lower = 1<<4,           // 小写字母
        alpha = 1<<5,           // 字母表字符
        digit = 1<<6,           // 十进制数字
        punct = 1<<7,           // 标点字符
        xdigit = 1<<8,          // 十六进制数字
        alnum=alpha|digit,      // 字母数字字符
        graph=alnum|punct
    };
};
```

这个 *mask* 并不依赖于特定的字符类型, 所以该枚举放在一个 (非模板的) 基类里。

很清楚, *mask* 反应了传统C和C++的字符类别 (20.4.1节)。当然, 对于不同的字符集, 落入不同类别中的字符则不同。例如, 对于ASCII字符集, 整数值125表示字符 '}', 这是一个标点字符 (*punct*); 然而在丹麦国家字符集里, 125表示元音 'å', 它在丹麦现场里必须分类为

一个`alpha`。

这种分类被称为“掩码”(mask)是因为对于小字符集的字符分类,传统的高效实现方式是用一个表,其中的每个表项保存着代表其类别的二进制位。例如,

```
table['a'] == lower | alpha | xdigit
table['1'] == digit
table[' '] == space
```

采用这种实现方式, `table[c] & m` 非零当且仅当 `c` 是一个 `m`, 否则值就是 0。

`ctype` 刻画面定义如下:

```
template <class Ch>
class std::ctype : public locale::facet, public ctype_base {
public:
    typedef Ch char_type;
    explicit ctype(size_t r = 0);

    bool is(mask m, Ch c) const; // c是m吗?

    // 将 [b:e) 中各个Ch的分类放进v:
    const Ch* is(const Ch* b, const Ch* e, mask* v) const;

    const Ch* scan_is(mask m, const Ch* b, const Ch* e) const; // 找出一个m
    const Ch* scan_not(mask m, const Ch* b, const Ch* e) const; // 找出一个非m

    Ch toupper(Ch c) const;
    const Ch* toupper(Ch* b, const Ch* e) const; // 转换 [b:e)
    Ch tolower(Ch c) const;
    const Ch* tolower(Ch* b, const Ch* e) const;

    Ch widen(char c) const;
    const char* widen(const char* b, const char* e, Ch* b2) const;
    char narrow(Ch c, char def) const;
    const Ch* narrow(const Ch* b, const Ch* e, char def, char* b2) const;

    static locale::id id; // 刻画标识符对象 (D.2节、D.3节、D.3.1节)

protected:
    ~ctype();

    // 为公用函数用的“do_”虚函数 (见D.4.1节)
};
```

调用 `is(m, c)` 检测字符 `c` 是否属于类别 `m`。例如,

```
int count_spaces(const string& s, const locale& loc)
{
    int i = 0;
    char ch;
    for(string::const_iterator p = s.begin(); p != s.end(); ++p)
        if (loc.is(space, ch)) ++i; // 由loc定义的空白
    return i;
}
```

注意,也可以用 `is()` 检查某个字符是否属于几个分类中的某一个。例如,

```
loc.is(ctype::space | ctype::punct, c); // c在loc里是空白或者标点吗?
```

调用 `is(b, e, v)` 确定 `[b:e)` 中各个字符的类别,并将结果放入数组 `v` 中的对应位置。

调用 `scan_is(m, b, e)` 返回一个指针,该指针指向 `[b:e)` 里第一个为 `m` 的字符。如果不存在类别为 `m` 的字符就返回 `e`。像其他标准刻画面一样,这些公用成员函数都是通过调用各自对应的

“do_”虚函数实现的。一个简单的实现可能是：

```
template <class Ch>
const Ch* std::ctype<Ch>::do_scan_is(mask m, const Ch* b, const Ch* e) const
{
    while (b!=e && !is(m, *b)) ++b;
    return b;
}
```

调用`scan_not(m, b, e)`返回一个指针，该指针指向**[b:e)**里第一个不是**m**的字符。如果所有字符都属于类别**m**就返回**e**。

如果字符集里有对应的大写形式的话，调用`toupper(c)`返回**c**的大写形式；否则就返回**c**本身。

调用`toupper(b, e)`将**[b:e)**里的所有字符转换为大写并返回**e**。其简单实现是：

```
template <class Ch>
const Ch* std::ctype<Ch>::to_upper(Ch* b, const Ch* e)
{
    for (; b!=e; ++b) *b = toupper(*b);
    return e;
}
```

`tolower()`函数与`toupper()`类似，只是转换到小写。

调用`widen(c)`将字符**c**转换到它对应的**Ch**值。如果**Ch**字符集提供了多个对应于**c**的字符，标准的描述是使用“最简单的合理转换”。例如，

```
wcout << use_facet<ctype<wchar_t>>(wcout.getloc()).widen('e');
```

将输出字符**e**在**wcout**的现场中的某个合理的对应字符。

无关的字符表示之间（例如在ASCII和EBCDIC之间）的转换也可以通过`widen()`完成。例如，假定存在一个**ebcdic**现场：

```
char EBCDIC_e = use_facet<ctype<char>>(ebcdic).widen('e');
```

调用`widen(b, e, v)`取出**[b:e)**中的每个字符，并将对其`widen()`后的版本放入数组**v**的对应位置里。

调用`narrow(ch, def)`产生一个与来自字符集**Ch**的**ch**对应的**char**值，同样使用“最简单的合理转换”。如果没有对应的**char**值就返回**def**。

调用`narrow(b, e, def, v)`取出**[b:e)**中的每个字符，并将对其`narrow()`后的版本放入数组**v**的对应位置里。

这里的一般性想法是，`narrow()`执行的是从一个大字符集到一个小字符集的转换，而`widen()`做与之相反的操作。对于来自小字符集的一个字符**c**，我们希望对

```
c == narrow(widen(c), 0) // 没有保证
```

如果由**c**表示的字符在“小字符集”里只有一个表示形式，那么上面等式就为真。当然，这并没有保证。如果一个**char**所代表的那些字符不是大字符集（**Ch**）所表示的字符的一个子集，我们在处理字符类别时就可能会遇到异常情况或者问题。

与此类似，对于大字符集里的字符**ch**，我们希望有

```
widen(narrow(ch, def)) == ch || widen(narrow(ch, def)) == widen(def) // 没有保证
```

虽然常常是这种情况，但是，对于一个字符在大字符集里有多个值而在小字符集里只有一个

值的情况，这个等式也不能保证。例如，数字7在大字符集里常常有多个表示形式。产生这种情况的典型原因是，大字符集包含几个规定的字符集作为子集，而小字符集里的字符在各个规定字符集里都有。

对于基本源字符集中的每个字符（C.3.3节），保证有

```
widen(narrow(ch_lit, 0)) == ch_lit
```

例如，

```
widen(narrow('x', 0)) == 'x'
```

`narrow()` 和 `widen()` 函数尽可能尊重字符分类情况。举个例子，假如 `is(alpha, c)`，那么 `is(alpha, narrow(c))` 且 `is(alpha, widen(c))`，这里的 `alpha` 是所用现场里的合法掩码。

一般情况下使用 `ctype` 剖面，而特殊情况下使用 `narrow()` 和 `widen()` 函数的一个主要原因就是：为了写出能够对任何字符集做 I/O 和字符串操作的代码，使它们对于字符集而言成为通用型代码。这也意味着 `iostream` 的实现密切地依赖于这些功能。通过依靠 `<iostream>` 和 `<string>`，用户就可以避免大部分对 `ctype` 剖面的直接使用了。

提供了 `ctype` 的一个 `_byname` 版本（D.4节、D.4.1节）：

```
template <class Ch> class std::ctype_byname : public ctype<Ch> { /* ... */ };
```

D.4.5.1 方便界面

`ctype` 剖面的一种最常见用法就是询问某个字符是否属于某个给定类别。为此提供了如下的一组函数：

```
template <class Ch> bool isspace(Ch c, const locale& loc);
template <class Ch> bool isprint(Ch c, const locale& loc);
template <class Ch> bool iscntrl(Ch c, const locale& loc);
template <class Ch> bool isupper(Ch c, const locale& loc);
template <class Ch> bool islower(Ch c, const locale& loc);
template <class Ch> bool isalpha(Ch c, const locale& loc);
template <class Ch> bool isdigit(Ch c, const locale& loc);
template <class Ch> bool ispunct(Ch c, const locale& loc);
template <class Ch> bool isxdigit(Ch c, const locale& loc);
template <class Ch> bool isalnum(Ch c, const locale& loc);
template <class Ch> bool isgraph(Ch c, const locale& loc);
```

这些函数很容易通过 `use_facet` 实现。例如：

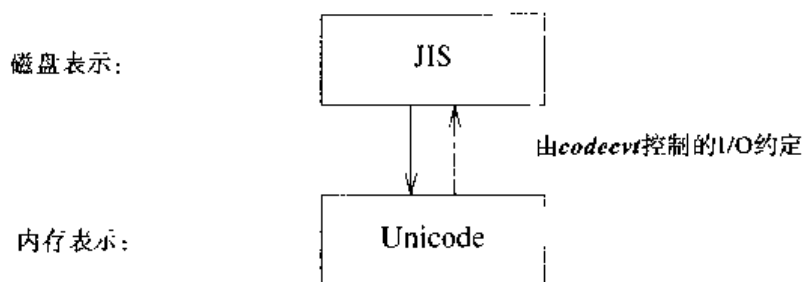
```
template <class Ch>
inline bool isspace(Ch c, const locale& loc)
{
    return use_facet<ctype<Ch>>(loc).is(space, c);
}
```

在20.4.2节给出了这些函数的单参数版本，它们就是上述函数针对当前C全局现场的版本（而不是相对于全局C++现场 `locale()` 的）。极特殊的情况中C全局现场与C++全局现场有差异（D.2.3节），除此之外我们可以认为单参数版本就是两参数版本对 `locale()` 的应用。例如，

```
inline int isspace(int i)
{
    return isspace(i, locale()); // 几乎都如此
}
```

D.4.6 字符编码转换

有时, 存储在文件里的字符表示形式与在内存中同样字符的表示不同。例如, 存于文件里的日文字符通常用指示符(“移位”)告知一个给定的字符序列属于四个常用字符集(日文汉字、片假名、平假名、日文罗马字)中的哪一个。这样做有点笨拙, 因为每个字节的意义将依赖于它的“移位状态”。但这种方式可以节约一些存储, 因为只有汉字需要用多个字节去表示。在内存中, 将这些字符都表示为多字节字符集将有利于操作, 这时的字符将具有同样的大小。这种字符(例如, Unicode字符)通常被存入宽字符中(`wchar_t`, 4.3节)。为此, 在刻画`codecvt`里提供一种机制, 用于在读入或者写出时将字符从一种表示形式转换到另一种形式。例如,



这种编码转换机制具有充分的通用性, 可以提供字符表示的任何转换。它使我们可以写出一个程序去使用某种合适的内部字符表示(存储在`char`、`wchar_t`或者其他类型里), 又可以通过`iostream`所用现场的调整去接受各种各样的输入流表示。完成此事的其他可能方式就是修改程序本身, 或者做各种输入/输出文件格式的转换。

`codecvt`刻画面所提供的不同字符集之间的转换, 是当字符在流缓冲区和外部存储之间转移时完成的:

```

class std::codecvt_base {
public:
    enum result { ok, partial, error, noconv };    // 结果指示符
};

template <class I, class E, class State>
class std::codecvt : public locale::facet, public codecvt_base {
public:
    typedef I intern_type;
    typedef E extern_type;
    typedef State state_type;

    explicit codecvt(size_t r = 0);

    result in(State&, const E* from, const E* from_end, const E*& from_next, // 读出
              I* to, I* to_end, I*& to_next) const;

    result out(State&, const I* from, const I* from_end, const I*& from_next, // 写出
              E* to, E* to_end, E*& to_next) const;

    result unshift(State&, E* to, E* to_end, E*& to_next) const; // 最终字符序列

    int encoding() const throw();    // 刻画基本编码特性
    bool always_noconv() const throw();    // 我们可以不做字符转换而I/O吗?

    int length(const State&, const E* from, const E* from_end, size_t max) const;
  
```

```

    int max_length() const throw();           // 最大的可能长度length()
    static locale::id id; // 刻面标识符对象 (D.2节、D.3节、D.3.1节)

protected:
    ~codecvt();

    // 为公用函数用的“do_”虚函数 (见D.4.1节)
};

```

codecvt刻面由**basic_filebuf** (21.5节)使用,用于读写字符。**basic_filebuf**从流的现场中得到这个刻面 (21.7.1节)。

这里的模板参数**State**是用于保存被转换的流的移位状态的类型。还可以用**State**去描述专门化,以便标识不同的转换。后一种用途很重要,因为一些具有不同字符编码(字符集)的字符有可能存储在同一个类型的对象里面。例如,

```

class JISstate { /* ... */ };

p = new codecvt<wchar_t, char, mbstate_t>; // 标准char到宽字符
q = new codecvt<wchar_t, char, JISstate>; // JIS到宽字符

```

如果没有不同的**State**参数,那么刻面可能就无法知道应要求**char**流采用何种编码方式。来自**<wchar>**或**<wchar.h>**的**mbstate_t**标识的是在**char**和**wchar_t**之间的标准转换。

也可以通过派生类创建新的**codecvt**,并通过名字标识。例如,

```

class JIScvt : public codecvt<wchar_t, char, mbstate_t> { /* ... */ };

```

调用**in(s, from, from_end, from_next, to, to_end, to_next)**从范围**[from, from_end)**读入每个字符并设法去转换它。如果能够转换,**in()**就将转换结果写进**[to, to_end)**范围里对应的位置;如果无法转换,**in()**就停止在那一点。在返回时,**in()**将超过所读结束处一个字符的位置存入**from_next**,将超过所写结束处一个字符的位置存入**to_next**。由**in()**返回的**result**值说明工作的情况:

ok	在 [from, from_end) 中的所有字符均已转换
partial	并不是 [from, from_end) 中的所有字符均已转换
error	in() 遇到了一个无法转换的字符
nonconv	不需要转换

注意,**partial**转换未必是错误,也可能还需要读入更多的字符才能读完一个多字节字符并将它写出去,或者需要腾空输出缓冲区为下面的字符腾出空间。

类型**State**的参数**s**指明了在对**in()**调用开始时输入字符序列的状态。如果外部字符表示采用了移位状态,这一点就非常重要。注意,**s**是一个(非**const**)引用参数:在调用结束时,**s**应保存着输入序列的移位状态。这就使程序员能够处理**partial**转换,以及通过对**in()**几次调用去转换一个长序列。

调用**out(s, from, from_end, from_next, to, to_end, to_next)**将范围**[from, from_end)**从内部表示转换到外部表示,其方式与**in()**从外部表示转换到内部表示一样。

一个字符序列必须从一个“中性”(无移位)的字符开始和结束。典型情况下,这个状态就是**State()**。调用**unshift(s, to, to_end, to_next)**查看**s**并根据需要将字符放进**[to, to_end)**,将字符序列转回未移位的状态。**unshift()**的结果和对**to_next**的使用与**out()**一样。

调用**length(s, from, from_end, max)**返回**in()**对**[from, from_end)**可能转换的字符数。

调用`encoding()`返回:

- 1 如果外部字符集的编码中使用了状态(例如,用移位和反移位字符序列)。
- 0 如果编码采用变动数目的字节表示单个字符(例如,一个字符表示可能采用一字节中的一个位指出表示该字符使用的是一个或者两个字节)。
- n* 如果外部字符表示的每个字符用*n*个字节。

如果在内部和外部字符集之间不需要转换,调用`always_noconv()`就返回`true`;否则就返回`false`。很清楚,`always_noconv() == true`为实现打开了一种可能性,使它可以提供不调用转换函数的最有效的实现。

调用`max_length()`返回`length()`对合法参数集合可能返回的最大值。

我能想到的最简单的编码转换就是转换到大写。这样,下面大概就是最简单的还能完成一点服务的`codecvt`:

```
class Cvt_to_upper : public codecvt<char, char, mbstate_t> {    // 转换为大写
    explicit Cvt_to_upper(size_t r = 0) : codecvt(r) {}

protected:
    // 读入外部表示, 写出内部表示;
    result do_in(State& s, const char* from, const char* from_end, const char*& from_next,
                char* to, char* to_end, char*& to_next) const;

    // 读入内部表示, 写出外部表示;
    result do_out(State& s, const char* from, const char* from_end, const char*& from_next,
                char* to, char* to_end, char*& to_next) const
    {
        return codecvt<char, char, mbstate_t>::do_out
            (s, from, from_end, from_next, to, to_end, to_next);
    }

    result do_unshift(State&, E* to, E* to_end, E*& to_next) const { return ok; }

    int do_encoding() const throw() { return 1; }
    bool do_always_noconv() const throw() { return false; }

    int do_length(const State&, const E* from, const E* from_end, size_t max) const;
    int do_max_length() const throw();    // 最大可能长度
};

codecvt<char, char, mbstate_t>::result
Cvt_to_upper::do_in(State& s, const char* from, const char* from_end,
                    const char*& from_next, char* to, char* to_end, char*& to_next) const
{
    // ...D.6[16] ...
}

int main()    // 简单测试
{
    locale ulocale(locale(), new Cvt_to_upper);
    cin.imbue(ulocale);
    char ch;
    while (cin >> ch) cout << ch;
}
```

提供了`_byname`(D.4节、D.4.1节)版本的`codecvt`:

```
template <class I, class E, class State>
class std::codecvt_byname : public codecvt<I, E, State> { /* ... */ };
```

D.4.7 消息

很自然，大部分用户最喜欢用他们自己的母语与程序交互；然而，我们却无法提供一种标准的机制去表达某种通过现场定制的一般性交互方式。标准库提供了一种简单机制，以保存一组由现场特别设置的字符串，供用户由它们组合出简单消息（message）。从本质上说，*messages*实现了一种简单的只读数据库：

```
class std::messages_base {
public:
    typedef int catalog; // 标识符类型的门类
};

template <class Ch>
class std::messages : public locale::facet, public messages_base {
public:
    typedef Ch char_type;
    typedef basic_string<Ch> string_type;

    explicit messages(size_t r = 0);

    catalog open(const basic_string<char>& fn, const locale&) const;
    string_type get(catalog c, int set, int msgid, const string_type& d) const;
    void close(catalog c) const;

    static locale::id id; // 刻画标识符对象（D.2节、D.3节、D.3.1节）

protected:
    ~messages();

    // 为公用函数用的do_虚函数（见D.4.1节）
};
```

调用`open(s, loc)`为现场`loc`打开一个称为`s`的消息“门类”。一个门类也就是一组字符串，以某种由实现确定的方式组织起来，可以通过`messages::get()`函数访问。如果无法打开名为`s`的消息门类，该函数返回负值。在第一次通过`get()`使用某个消息门类之前必须先打开它。

调用`close(cat)`关闭由`cat`标明的消息门类，并释放与该门类关联的资源。

调用`get(cat, set, id, "foo")`在消息门类`cat`里查询由`(set, id)`标识的消息。如果找到一个字符串，`get()`就返回它；否则`get()`返回默认的串（这里是`string("foo")`）。

下面是为某个实现所用的一个*messages*刻画示例，其中的消息门类是“消息”的集合的向量，而一个“消息”就是一个字符串：

```
struct Set {
    vector<string> msgs;
};

struct Cat {
    vector<Set> sets;
};

class My_messages : public messages<char> {
    vector<Cat>& catalogs;
public:
    explicit My_messages(size_t = 0) : catalogs(*new vector<Cat>) { }

    catalog do_open(const string& s, const locale& loc) const; // 打开消息门类s
    string do_get(catalog c, int s, int m, const string&) const; // 从c中取消息(s, m)
    void do_close(catalog cat) const;
```

```

    {
        if (catalogs.size() <= cat) catalogs.erase(catalogs.begin() + cat);
    }
    ~My_messages() { delete &catalogs; }
};

```

`messages`的所有成员函数都是`const`，所以其消息门类数据结构(`vector<set>`)存在刻面之外。

选择消息是通过指定一个消息门类、该门类中的一个集合和该集合中的一个字符串的方式来完成。提供了一个字符串参数，用于在该门类中找不到消息的情况下作为默认结果：

```

string My_messages::do_get(catalog cat, int set, int msg, const string& def) const
{
    if (catalogs.size() <= cat) return def;
    Cat& c = catalogs[cat];
    if (c.sets.size() <= set) return def;
    Set& s = c.sets[set];
    if (s.msgs.size() <= msg) return def;
    return s.msgs[msg];
}

```

打开一个消息门类涉及到从磁盘读一个文本表示到`Cat`结构。在这里，我选择的是一种很容易读的表示方式。一个集合用<<<和>>>作为分界，每个消息是一个正文行：

```

messages<char>::catalog My_messages::do_open(const string& n, const locale& loc) const
{
    string nn = n + locale().name();
    ifstream f(nn.c_str());
    if (!f) return -1;

    catalogs.push_back(Cat()); // 做出在内存的消息门类
    Cat& c = catalogs.back();
    string s;
    while (f>>s && s!="<<<") { // 读Set
        c.sets.push_back(Set());
        Set& ss = c.sets.back();
        while (getline(f, s) && s!=">>>") ss.msgs.push_back(s); // 读消息
    }
    return catalogs.size() - 1;
}

```

下面是一个简单应用：

```

int main()
{
    if (!has_facet<My_messages>(locale())) {
        cerr << "no messages facet found in " << locale().name() << '\n';
        exit(1);
    }

    const messages<char>& m = use_facet<My_messages>(locale());
    extern string message_directory; // 我保存消息的地方
    int cat = m.open(message_directory, locale());
    if (cat < 0) {
        cerr << "no catalog found\n";
        exit(1);
    }

    cout << m.get(cat, 0, 0, "Missed again!") << endl;
    cout << m.get(cat, 1, 2, "Missed again!") << endl;
}

```

```

        cout << m.get(cat, 1, 3, "Missed again!") << endl;
        cout << m.get(cat, 3, 0, "Missed again!") << endl;
    }

```

如果消息门类是

```

<<<
hello
goodbye
>>>
<<<
yes
no
maybe
>>>

```

这个程序将打印

```

hello
maybe
Missed again!
Missed again!

```

D.4.7.1 使用来自其他刻面的消息

除了在与用户的交互中作为一种与现场无关的展台之外，消息还可以为其他剖面保存字符串。例如，**Season_io**剖面（D.3.2节）也可以写成下面的样子：

```

class Season_io : public locale::facet {
    const messages<char>& m;      // 消息字典
    int cat;                      // 消息门类
public:
    class Missing_messages { };

    Season_io(int i = 0)
        : locale::facet(i),
          m(use_facet<Season_messages>(locale())),
          cat(m.open(message_directory, locale()))
    { if (cat < 0) throw Missing_messages(); }

    ~Season_io() { } // 使Season_io对象能够销毁（D.3节）

    const string& to_str(Season x) const;          // x的字符串表示
    bool from_str(const string& s, Season& x) const; // 在x里放置对应s的Season
    static locale::id id; // 剖面标识符对象（D.2节、D.3节、D.3.1节）
};

locale::id Season_io::id; // 定义标识符对象
const string& Season_io::to_str(Season x) const
{
    return m->get(cat, x, "no-such-season");
}

bool Season_io::from_str(const string& s, Season& x) const
{
    for (int i = Season::spring; i <= Season::winter; i++)
        if (m->get(cat, i, "no-such-season") == s) {
            x = Season(i);
            return true;
        }
}

```



```
    return false;
}
```

这种基于消息的解决方案与原来解法（D.3.2节）的不同之处在于：如果实现者要为新现场提供一组Season串，那么就可以将它们加入messages字典中。这将使人们更容易向一个执行环境里增加新的现场。当然，因为messages提供的是只读界面，增加一组新赛季名的工作可能已经超出了应用程序员的工作范围。

同样提供了_byname（D.4节、D.4.1节）版本的messages：

```
template <class Ch>
class std::messages_byname : public messages<Ch> { /* ... */};
```

D.5 忠告

- [1] 应预期每个直接与人打交道的非平凡程序或者系统都会用在多个国家；D.1节。
- [2] 不要假定每个人使用的都是你所用的字符集；D.4.1节。
- [3] 最好是用locale而不是写实质性代码去做对文化敏感的I/O；D.1节。
- [4] 避免将现场名字字符串嵌入到程序正文里；D.2.1节。
- [5] 尽可能减少全局格式信息的使用；D.2.3节、D.4.4.7节。
- [6] 最好是用与现场有关的字符串比较和排序；D.2.4节、D.4.1节。
- [7] 保存facet的不变性；D.2.2节、D.3节。
- [8] 应保证改变现场的情况只出现在程序里的几个地方；D.2.3节。
- [9] 利用现场去管理刻面的生存期；D.3节。
- [10] 在写对现场敏感的I/O函数时，记住去处理用户（通过覆盖）提供的函数所抛出的异常；D.4.2.2节。
- [11] 用简单的Money类型保存货币值；D.4.3节。
- [12] 要做对现场敏感的I/O，最好是用简单的用户定义类型保存所需的值（而不是从内部类型的值强制转换）；D.4.3节。
- [13] 在你涉及到所有因素有了很好的看法之前，不要相信计时结果；D.4.4.1节。
- [14] 当心time_t的取值范围；D.4.4.1节、D.4.4.5节。
- [15] 使用能接受多种输入格式的日期输入例程；D.4.4.5节。
- [16] 最好采用那些明显表明了所用现场的字符分类函数；D.4.5节、D.4.5.1节。

D.6 练习

1. (*2.5) 为某种不同于美国英语的语言定义一个Season_io（D.3.2节）；
2. (*2) 定义一个Season_io（D.3.2节）类，它以一组名字串作为一个构造函数的参数，以使不同现场的Season名字都可以表示为这个类的对象。
3. (*3) 写一个给出字典序的collate<char>::compare()。最好是针对另一种语言（如德语或者法语）去做，该语言字母表里的字母比英语多。
4. (*2) 写一个程序，它采用数值的方式、英文词的方式和你所选定的其他语言的词的方式读入bool值。
5. (*2.5) 定义一个Time类型来表示一天中的时间。用Date类型和Time类型定义出一个

*Date_and_Time*类型。与D.4.4节的*Date*比较,讨论这种方式的优点和缺点。为*Time*和*Date_and_Time*实现与现场有关的I/O。

6. (*2.5) 设计和实现一个邮政编码刻画,为至少两个采用不同的写地址方式的国家实现它。例如:*NJ 07932*和*CB21QA*。
7. (*2.5) 设计和实现一个电话号码刻画,为至少两个采用不同的写电话号码习惯的国家实现它。例如:*(973) 360-8000*和*1223 343000*。
8. (*2.5) 做试验去确定你使用的那个实现对日期信息所用的输入和输出格式。
9. (*2.5) 定义一个*get_time()*,它能设法“猜测”具有歧义性的日期信息的意义,如12 5 1995,但仍能拒绝所有的或者几乎所有的错误。弄清楚什么样的“猜测”是可接受的,并讨论出错的可能性。
10. (*2) 定义一个*get_time()*,使它能接受比D.4.4.5节的那个函数变化更多的输入格式。
11. (*2) 给出一个你的系统所支持的现场的列表。
12. (*2.5) 找出在你的系统里命名的现场存放的位置。如果你有权访问系统中存放现场的地方,请做出一个新现场。当心不要破坏了已有的现场。
13. (*2) 比较*Season_io*的两个实现(D.3.2节和D.4.7.1节)。
14. (*2) 写出并测试一个*Date_out*刻画,它使用以构造函数的参数的形式提供的格式写*Date*。讨论这种方式与通过*date_fmt*提供的全局数据格式方式(D.4.4.6节)相比的优点和缺点。
15. (*2.5) 实现罗马数值(例如*XI*和*MDCLII*)的I/O。
16. (*2.5) 实现并测试*Cvt_to_upper*(D.4.6节)。
17. (*2.5) 用*clock()*确定下面操作的平均代价:(1)一次函数调用,(2)一次虚函数调用;(3)读入一个字符;(4)读入一个包含一个数字的*int*;(5)读入一个包含5个数字的*int*;(6)读入一个包含5个数字的*double*;(7)读入一个包含一个字符的*string*;(8)读入一个包含5个字符的*string*;(9)读入一个包含40个字符的*string*。
18. (*6.5) 学习另一种自然语言。

附录E 标准库的异常时安全性

一切将如你所料，
除非你的预期不对。
——Hyman Rosen

异常时安全性——异常时安全性的实现技术——资源的表示——赋值——*push_back()*
——构造函数和不变式——标准容器的保证——元素的插入和删除——保证和权衡——
swap()——初始化和迭代器——对元素的引用——谓词——*string*、流、算法、*valarray*
和*complex*——C标准库——对库用户的寓意——忠告——练习

E.1 引言

标准库函数常需要去调用一些用户以函数或者模板参数的形式提供的操作。很自然，用户所提供的某些操作偶尔会抛出异常。另外一些函数，如分配器函数，也可能抛出异常。考虑

```
void f(vector<X>& v, const X& g)
{
    v[2] = g;                // X的赋值可能抛出异常
    v.push_back(g);          // vector<X>的分配器可能抛出异常
    sort(v.begin(), v.end()); // X的小于操作可能抛出异常
    vector<X> u = v;          // X的复制构造函数可能抛出异常
    // ...
    // u在这里析构：我们必须保证X的析构函数能正确工作
}
```

如果在试图复制 g 时赋值操作抛出异常会出现什么情况呢？会给 v 留下一个非法元素吗？如果 $v.push_back()$ 用于去复制 g 的构造函数抛出`std::bad_alloc`，那么又会发生什么情况？元素的个数会改变吗？会不会把一个非法元素加进容器里？如果在排序过程中 X 的小于运算符抛出异常会发生什么情况？元素会被部分排序吗？会不会有某个元素被排序算法从容器里删除而没有放回去？

列举出在这个例子里所有可能异常的工作留做练习（E.8[1]）。解释清楚这个例子对于定义完好的 X 为什么有良好的行为（即使是某个 X 抛出了异常）则是本附录的一部分目标。很自然，这个解释的主要部分将涉及到，在异常的环境中给出“行为良好”和“定义良好”概念的确切含义和一些有效的术语。

本附录的目的是：

- [1] 指明用户应该如何设计类型，以便能满足标准库的要求。
- [2] 说明由标准库所提供的保证。
- [3] 说明标准库对用户代码的要求。
- [4] 展示一些构造具有异常时安全性的高效容器的有效技术。

[5] 给出有关异常时安全性程序设计的若干普遍性规则。

有关异常时安全性的讨论必须集中关注最坏情况下的行为，也就是说，在哪里发生异常将导致最严重的问题？标准库如何针对潜在的问题保护自己及其用户？还有，用户能如何提供帮助以防止出现问题？请不要让这一有关异常处理技术的讨论分散了你对一个核心事实的认识，那就是：抛出异常是报告错误的最好方法（14.1节、14.9节）。有关概念、技术和标准库所提供保证的讨论将按如下方式组织：

E.2节 讨论异常时安全性的概念。

E.3节 介绍实现有效的具有异常时安全性的容器及其操作的技术。

E.4节 描述标准库容器及其操作所提供的保证。

E.5节 综述标准库的非容器部分与异常时安全性有关的问题。

E.6节 从标准库用户的观点重新观察异常时安全性的有关问题。

与其他地方一样，标准库为在应用中必须考虑和处理的这方面问题提供了许多示例。用于为标准库提供异常时安全性的技术可以用于范围广泛的许多问题。

E.2 异常时安全性

对某个对象的某个操作称为具有异常时安全性，如果当这个操作由于抛出异常而结束时，该操作能将这个对象留在一个合法状态中。这个合法状态可以是一个要求进一步清理的错误状态，但它必须是定义良好的，使人们可以为此对象写出合理的错误处理代码。例如，一个异常处理器可能去销毁这个对象，修复这个对象，重复做原来操作的某种变形，简单地继续做下去，如此等等。

换句话说，这个对象将具有某种不变式（24.3.7.1节），它的构造函数建立起这个不变式；所有进一步执行的操作都能维持这个不变式，即使是遇到抛出异常的情况；它的析构函数将完成最后的清理。一个操作在抛出异常之前，应该关心不变式的维持问题，使有关的对象处于合法的状态。当然，也有可能出现这种合法状态并不符合应用需要的情况。例如，在一个串里留下了一个空串或者容器可能还是没有排好序。这样，“修复”也就意味着给对象一个值，该值比操作失败后留下的值更适合或者更符合应用的需要。在考虑与标准库有关的问题时，最有趣的对象就是容器。

在这里我们要考虑，在什么条件下标准库容器可以被认为是异常时安全的。可能存在两种概念上相当简单的策略：

[1] “没有任何保证”：如果抛出了异常，任何正在操作中的容器都可能毁坏。

[2] “强保证”：如果抛出异常，任何正在操作中的容器都维持在这个标准库操作开始前所处的那个状态。

不幸的是，这两种回答都过于简单，以至无法实用。选择[1]是不能接受的，因为它意味着一旦从某个容器操作里抛出异常之后，该容器就不可被访问了，为了不致引起运行时错误，我们甚至不能去销毁它。选择[2]也是不能接受的，因为这将对每个标准库操作强求一个代价高昂的可恢复式的语义。

为了解决这种两难问题，C++标准库提供了一组异常时安全性的保证，它们将做出正确程序的重担分摊到标准库的实现者和标准库的用户两者身上：

[3a] “对于所有操作的基本保证”：保持标准库的基本不变式，而且不会出现资源（例如

存储)的流失。

[3b] “对于关键操作的强保证”：除了提供基本保证外，这些操作或者成功、或者毫无影响。只对一些关键性操作提供了这种保证，例如`push_back()`，对于`list`的单个元素`insert()`以及`uninitialized_copy()` (E.3.1节、E.4.1节)。

[3c] “某些操作的不抛出”：除了提供基本保证之外，有些操作保证不会抛出异常。对某些简单操作提供了这一保证，例如`swap()`和`pop_back()` (E.4.1节)。

提供基本保证和强保证都有条件，这里要求用户所提供的操作（例如赋值和`swap()`函数）不会将容器元素留在非法状态中，以及用户提供的操作不会引起资源流失，还有就是析构函数不抛出异常。举个例子，考虑下面这些“类似句柄” (25.7节)的类：

```
template<class T> class Safe {
    T* p;      // p指向一个通过new分配的T
public:
    Safe() : p(new T) { }
    ~Safe() { delete p; }
    Safe& operator=(const Safe& a) { *p = *a.p; return *this; }
    // ...
};

template<class T> class Unsafe {      // 散漫而危险的代码
    T* p;      // p指向一个T
public:
    Unsafe(T* pp) : p(pp) { }
    ~Unsafe() { if (!p->destructible()) throw E(); delete p; }
    Unsafe& operator=(const Unsafe& a)
    {
        p->~T();          // 销毁老值 (10.4.11节)
        new(p) T(a.p);    // 在 *p构造起a.p的副本 (10.4.1节)
        return *this;
    }
    // ...
};

void f(vector< Safe<Some_type> >&vg, vector< Unsafe<Some_type> >&vb)
{
    vg.at(1) = Safe<Some_type>();
    vb.at(1) = Unsafe<Some_type>(new Some_type);
    // ...
}
```

在这个例子里，只要`T`的构造成功，`Safe`的构造就一定成功。对`T`的构造可能失败，因为分配失败（并抛出`std::bad_alloc`），或是因为`T`的构造函数抛出了异常。然而，对于每个成功创建的`Safe`，`p`都指向一个成功创建的`T`对象；如果一个构造函数失败，那么就没有创建出`T`对象（也没创建`Safe`对象）。与此类似，`T`的赋值运算符可能抛出异常，并导致`Safe`的赋值运算符又重新抛出这个异常。然而，只要`T`的赋值运算符总将其运算对象留在良好的状态，这样也不会有问题。因此，`Safe`是行为良好的，随之，对于`Safe`的每个标准库操作都将得到合理的良好定义的结果。

另一方面，`Unsafe()`的书写则非常马虎（或者说，它是很仔细地写出的，目的就是展示这些不希望有的行为）。`Unsafe`的构造不会失败。与此相应的是，`Unsafe`的各种操作（例如赋值和析构）被放到了一种必须处理各种潜在问题的情景之中。赋值运算符可能由于`T`的构造函

数抛出异常而失败，这将使那个 T 对象处于一种无定义状态，因为 $*p$ 的老值已被销毁，然而又没有新值去取代它。一般来说，这样做的后果是无法预料的。*Unsafe*的析构函数包含一种以病态形式表达的想法，目的是防止某种不应该做的析构。然而，在异常处理的过程中抛出一个异常将导致调用`terminate()` (14.7节)，标准库要求析构函数在销毁了对象之后正常返回。当用户提供的对象具有这种糟糕行为时，标准库没有（也不可能）提供任何保证。

从异常处理的观点看，*Safe*和*Unsafe*的差异在于：*Safe*用它的构造函数建立起一个不变式 (24.3.7.1节)，使它的操作能够简单而安全地实现。如果无法建立这个不变式，那么就抛出异常，它不会去创建不合法的对象。而在另一方面，*Unsafe*糊里糊涂没有一个有意义的不变式，操作各自抛出异常而没有一个整体的错误处理策略。自然，这样做的结果违反了标准库有关一个类型的行为的（合理）假设。举例说，*Unsafe*可能在 $T::operator=()$ 抛出异常之后在容器里留下一个非法元素，它的析构函数也可能抛出异常。

请注意，标准库的保证与用户所提供的具有病态行为的操作的关系同语言与违反基本类型系统的关系很类似。如果对某个基本功能的使用不符合它的规范，所产生的行为就没有定义。举个例子，如果你从一个`vector`元素的析构函数里抛出了一个异常，那就不要期望能够得到比间接访问一个用随机数初始化的指针更合理的结果：

```
class Bomb {
public:
    // ...
    ~Bomb() { throw Trouble(); };
};

vector<Bomb> b(10); // 导致无定义行为

void f()
{
    int* p = reinterpret_cast<int*>(rand()); // 导致无定义行为
    *p = 7;
}
```

用正面的话说：如果你遵循语言和标准库的基本规则，即使你抛出了异常，标准库也能有很好的行为。

除了达到纯粹的异常时安全性之外，我们通常还希望避免资源流失。也就是说，一个抛出了异常的操作不仅应该将它的操作对象都留在定义良好的状态中，还应该保证它已经申请的所有资源（最终）都能够释放掉。举例来说，在抛出异常的那个点，已经分配的所有存储必须被释放或者由某个对象所拥有的，而这个对象又能保证正确地释放这些存储。

标准库保证：只要所调用的由用户提供的操作能够避免资源流失，那么就不会出现资源流失。考虑

```
void leak(bool abort)
{
    vector<int> v(10); // 没有流失
    vector<int>* p = new vector<int>(10); // 可能出现存储流失
    auto_ptr<vector<int>> q(new vector<int>(10)); // 没有流失 (14.4.2节)

    if (abort) throw Up();
    // ...
    delete p;
}
```

在抛出异常时，称为 v 的`vector`和由 q 保持的`vector`都将被正确销毁，它们的资源将被释放。由 p 所指的向量则没有针对异常的任何保证，也不会被销毁。为了使这段代码安全，我们在抛出异常之前应该显式地销毁 p 或者保证它被某个对象（例如一个`auto_ptr`（14.4.2节））所拥有，如果抛出异常，那个对象将正确地销毁它。

注意，语言中关于部分构造和析构的规则保证，在构造子对象和成员期间抛出的异常都能得到正确的处理，无须标准库代码的特别关注（14.4.1节）。这一规则是所有处理异常的技术最根本的支撑点。

还有，应该记得存储并不是惟一的一种可能流失的资源。打开的文件、锁、网络连接和线程等也都是系统资源的示例，一个函数在抛出异常之前必须释放它们，或者将它们转交给某个对象。

E.3 异常时安全性的实现技术

与其他地方一样，标准库提供了许多问题示例，这些问题也出现在其他许多环境里，其解决方案能够广泛应用。写出具有异常时安全性的代码的基本工具包括：

[1] `try`块（8.3.1节）。

[2] 对于“资源申请即初始化”技术的支持（14.4节）。

应遵循的普遍性原则是：

[3] 在我们有了能存入某信息片段的替代物之前，绝不要将原片段丢掉。

[4] 在抛出和重新抛出异常时，总将对象留在合法的状态中。

按此方式，我们就总能从错误境况中全身而退。坚持这些原则的实际困难在于一些看起来无害的操作（例如`<`、`=`和`sort()`）也可能抛出异常。要理解应该到一个应用里的什么地方去查看需要做试验。

在你写一个库的时候，理想应是将目标定在强的异常时安全性保证上（E.2节），并始终提供基本保证。在写一个特殊程序时，异常时安全性的问题就可能少一些。举例说，如果我写一个自己用的简单数据分析程序，在发生了不大可能的事件，如虚拟存储耗尽，我通常会希望让程序停下来。不过，正确性和基本的异常时安全性是密切相关的。

提供基本的异常时安全性的各种技术，例如定义和检查不变式（24.3.7.1节），与那些对于保持程序比较小而且正确的极其有用的技术很类似。因此，为提供基本的异常时安全性（基本保证，E.2节）或者强保证所付出的额外代价有可能减到很小，甚至根本就微不足道；见E.8[17]。

在这里，我将考虑标准容器`vector`（16.3节）的一个实现，看看为了达到理想，它应该做些什么，以及在哪些地方我们更希望更受限的安全性。

E.3.1 一个简单的向量

`vector`的典型实现由一个句柄组成，句柄保存着到向量的第一个元素的指针、到超过最后元素一个位置的指针和到超过所分配空间一个元素的指针（17.1.3节）（或者等价地，采用一个指针加上两个偏移量）：



这里是`vector`的一个简化的声明，其中只给出了讨论异常时安全性和避免资源流失所需要的部分：

```
template<class T, class A = allocator<T> > class vector {
public:
    T* v;          // 所分配空间的开始
    T* space;       // 元素序列的结束，已分配供可能扩充的空间开始
    T* last;        // 所分配空间的结束
    A alloc;        // 分配器

    explicit vector(size_type n, const T& val = T(), const A& = A());

    vector(const vector& a);          // 复制构造函数
    vector& operator=(const vector& a); // 复制赋值

    ~vector();

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);

    // ...
};
```

首先考虑构造函数的一个朴素实现：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a) // 警告：朴素实现
    :alloc(a)                                              // 复制分配器
{
    v = alloc.allocate(n);                                // 为元素取得存储（19.4.1节）
    space = last = v+n;
    for (T* p = v; p!=last; ++p) a.construct(p, val);    // 在*p里构造val的副本（19.4.1节）
}
```

在这里存在着三个异常的来源：

- [1] `allocate()` 抛出异常，说明没有可用的空间。
- [2] 分配器的复制构造函数抛出异常。
- [3] 元素类型`T`的复制构造函数抛出异常，说明它无法复制`val`。

在这几种情况下，有关的对象都没有创建，因此不会调用`vector`的析构函数（14.4.1节）。

当`allocate()`失败时，`throw`将导致在任何分配之前退出，一切都很好。

当`T`的复制构造函数失败时，我们已经获得了一些空间，因此就必须释放它们，以避免存储流失。还有一个更困难的问题，`T`的复制构造函数可能在已经成功地创建了几个元素，但是在没有将它们全部创建起来之前抛出一个异常。

为了处置这一问题，我们可以保存好哪些元素已经构造完成的轨迹，并在出现错误时销毁它们（并且仅销毁它们）：

```
template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a) // 精细化的实现
    :alloc(a)                                              // 复制分配器
{
    v = alloc.allocate(n);                                // 为元素取得存储

    iterator p;

    try {
```



```

        iterator end = v+n;
        for (p=v; p!=end; ++p) alloc.construct(p, val);    // 构造元素 (19.4.1节)
        last = space = p;
    }
    catch (...) {
        for (iterator q = v; q!=p; ++q) alloc.destroy(q);    // 销毁已经构造的元素
        alloc.deallocate(v, n);    // 释放存储
        throw;    // 重新抛出
    }
}

```

这里的额外开销就是try块的开销。在很好的C++实现里，与分配存储和初始化元素相比，这一开销完全是微不足道的。对于那些进入try块引起很大代价的实现，可能值得在try之前加一个if(n)检查，另行处理空向量的情况。

这一构造函数的主要部分是一个uninitialized_fill()的异常时安全的实现：

```

template<class For, class T>
void uninitialized_fill(For beg, For end, const T& x)
{
    For p;
    try {
        for (p=beg; p!=end; ++p)
            new (static_cast<void*>(&*p)) T(x);    // 在*p中构造x的副本 (10.4.11节)
    }
    catch (...) { // 销毁已构造的元素并重新抛出;
        for (For q = beg; q!=p; ++q) (&*q)->~T(); // (10.4.11节)
        throw;
    }
}

```

这里的奇怪结构&*p是为了处理那些非指针的迭代器，在那种情况下，我们就需要通过间接得到所需的指针值，取得元素的地址。显式强制到void* 保证了对标准库放置函数的使用 (19.4.5节)，而不会去用用户为T定义的某个new。这一代码在相当低的层次上操作，在那个层次中，写出真正通用的代码可能很困难。

幸好，我们并不需要重新实现uninitialized_fill()，因为标准库为它提供了所需要的强保证 (E.2节)。初始化操作通常应该有一些最本质的性质：或者成功完成并初始化了每个元素；或者失败，但不会留下任何构造好的元素。因此，标准库算法uninitialized_fill()、uninitialized_fill_n()和uninitialized_copy() (19.4.1节)都保证具有这种强的异常时安全性质 (E.4.4节)。

注意，uninitialized_fill()算法没能力去抵御由元素的析构函数或迭代器操作 (E.4.4节)抛出的异常，要想这样做将带来无法接受的昂贵代价 (见E.8[16~17])。

uninitialized_fill()算法可以应用于多种不同的序列，因此它使用了一个前向迭代器 (19.2.1节)，且不能保证按照元素构造的相反顺序去销毁它们。

利用uninitialized_fill()，我们可以写出

```

template<class T, class A>
vector<T, A>::vector(size_type n, const T& val, const A& a)    // 糟糕的实现
: alloc(a)    // 复制分配器
{
    v = alloc.allocate(n);    // 为元素取得存储
    try {

```

```

        uninitialized_fill(v, v+n, val); // 复制元素
        space = last = v+n;
    }
    catch (...) {
        alloc.deallocate(v, n); // 释放存储
        throw; // 重新抛出
    }
}

```

然而我不能说这是好代码，下一节将阐述如何将它做得更简单些。

注意，构造函数重新抛出了所捕获的异常。这样做的意图就是让**vector**具有对异常的透明性，使用户可以确定究竟什么是问题的起因。所有标准库容器都具有这种性质。对异常的透明性通常是模板和其他“薄的”软件层应该采用的最佳策略。这与系统的那些主要部分（“模块”）形成鲜明对比，在那里需要对抛出的异常负起责任。也就是说，那样一个模块的实现者必须列出该模板可能抛出的每个异常。为做到这件事，可能涉及到异常的分组（14.2节），将低层次例行程序的异常映射到模块本身的异常（14.6.3节）或者异常的专门化（14.6节）等。

E.3.2 显式地表示存储

经验说明，通过显式使用try块的方式写出正确的异常时安全代码比大部分人想像的要困难许多。事实上，这种困难并不必要，因为存在着一种替代方式：“资源申请即初始化”技术（14.4节），它可以用于减少所需写出的代码量，并使这些代码更加规范。在目前情况下，**vector**所需要的关键性资源就是保存其元素的存储。通过提供一个辅助类来表示**vector**所用存储的概念，我们就可以简化代码，并减少无意间忘记释放存储的可能性：

```

template<class T, class A = allocator<T> >
struct vector_base {
    A alloc; // 分配器
    T* v; // 所分配空间的开始
    T* space; // 元素序列结束，已分配供可能扩充的空间开始
    T* last; // 所分配空间的结束

    vector_base(const A& a, typename A::size_type n)
        : alloc(a), v(a.allocate(n)), space(v+n), last(v+n) {}
    ~vector_base() { alloc.deallocate(v, last-v); }
};

```

只要**v**和**last**正确，**vector_base**就能销毁。类**vector_base**处理的是类型**T**的存储，而不是类型**T**的对象。因此，**vector_base**的用户就必须在销毁某个**vector_base**本身之前，先行销毁位于这个**vector_base**里的所有已构造好的对象。

很自然，**vector_base**的写法能保证：如果抛出了异常（由分配器的复制构造函数或者由**allocate()**函数），将不会创建起**vector_base**对象，也不会有存储流失现象。

有了**vector_base**之后，**vector**就可以定义如下：

```

template<class T, class A = allocator<T> >
class vector : private vector_base<T, A> {
    void destroy_elements() { for (T* p = v; p != space; ++p) p->~T(); space = v; } // 10.4.11节
public:
    explicit vector(size_type n, const T& val = T(), const A& a = A());

    vector(const vector& a); // 复制构造函数
    vector& operator=(const vector& a); // 复制赋值

```

```

    ~vector() { destroy_elements(); }

    size_type size() const { return space-v; }
    size_type capacity() const { return last-v; }

    void push_back(const T&);

    // ...
};

```

vector的析构函数显式地对每个元素调用**T**的析构函数。这也意味着，如果某个元素的析构函数抛出一个异常，**vector**的析构就会失败。此事如果发生在由异常导致的堆栈回退过程中，就会成为一个大灾难，并导致调用**terminate()**（14.7节）。在正常的析构中，从析构函数抛出的异常通常将引起资源的流失，并导致那些依赖于对象正常行为的代码出现无法预料的行为。不存在任何真正好的方式去防止由析构函数抛出的异常，因此，标准库对于元素析构函数的抛出动作没有任何保证（E.4节）。

现在，构造函数的定义变得很简单了：

```

template<class T, class A>
vector<T,A>::vector(size_type n, const T& val, const A& a)
    :vector_base(a,n)           // 为n个元素分配存储
{
    uninitialized_fill(v,v+n,val); // 复制元素
}

```

复制构造函数的不同点就在于用的是**uninitialized_copy()**而不是**uninitialized_fill()**：

```

template<class T, class A>
vector<T,A>::vector(const vector<T,A>& a)
    :vector_base(a,a.size())
{
    uninitialized_copy(a.begin(),a.end(),v);
}

```

注意，这种风格的构造函数依赖于语言的基础规则：在一个构造函数里抛出异常时，已经完全构造好的子对象（例如基类对象）将被正确销毁（14.4.1节）。**uninitialized_fill()**及其兄弟算法（E.4.4节）为部分构造好的序列提供了类似的保证。

E.3.3 赋值

与平常一样，赋值与构造的不同点在于必须关心原来的值。考虑下面的直接实现：

```

template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a) // 提供强保证（E.2节）
{
    vector_base<T,A> b(alloc,a.size());           // 取得存储
    uninitialized_copy(a.begin(),a.end(),b.v);     // 复制元素
    destroy_elements();                             // 释放原有存储
    alloc.deallocate(v,last-v);                     // 设置新表示
    vector_base::operator=(b);                     // 防止释放
    b.v = 0;
    return *this;
}

```

这个赋值很好，也是异常时安全的；但是它重复执行了许多构造函数和析构函数的代码。为了避免这种情况，我们可以写

```

template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a) // 提供强保证 (E.2节)
{
    vector temp(a); // 复制a
    swap<vector_base<T,A>>>(*this, temp); // 交换表示
    return *this;
}

```

老的元素由`temp`的析构函数销毁，用于保存它们的存储空间由`temp`中`vector_base`的析构函数释放。

这两个版本的性能也应该等价。从本质上说，它们两者所描述的不过是同一组操作的两种不同方式。然而，第二个实现更短一些，并且无须重复地写与`vector`函数相关的代码，所以，按这种方式写赋值应该更不容易出错，维护起来也更简单些。

应注意到这里缺少了传统的有关自我赋值的测试 (10.4.4节)：

```
if (this == &a) return *this;
```

这种赋值实现的工作方式是首先构造出一个副本，而后交换表示。这一做法明显可以正确处理自我赋值的情况。我认为，与给不同`vector`之间赋值的常见情况带来的额外代价相比，测试罕见的自我赋值情况所取得的效率收获并不合算。

上面两种实现中都忽视了两项潜在的重要优化：

[1] 如果被赋值向量的容量足够大，足以存放被赋值的那个向量，我们就不必去分配新存储。

[2] 一次元素赋值可能比析构一个元素，随后再构造一个元素的效率更高些。

实现了这些优化之后，我们得到

```

template<class T, class A>
vector<T,A>& vector<T,A>::operator=(const vector& a) // 优化后，基本保证 (E.2节)
{
    if (capacity() < a.size()) { // 分配新项的表示
        vector temp(a); // 复制a
        swap<vector_base<T,A>>>(*this, temp); // 交换表示
        return *this;
    }

    if (this == &a) return *this; // 防御自我赋值 (10.4.4节)

    // 赋值已有的元素
    size_type sz = size();
    size_type asz = a.size();
    alloc = a.get_allocator(); // 复制分配器
    if (asz <= sz) {
        copy(a.begin(), a.begin() + asz, v);
        for (T* p = v + asz; p != space; ++p) p->~T(); // 销毁过剩的元素
    }
    else {
        copy(a.begin(), a.begin() + sz, v);
        uninitialized_copy(a.begin() + sz, a.end(), space); // 创建额外的元素
    }
    space = v + asz;
    return *this;
}

```

这种优化并非没有代价。复制算法`copy()` (18.6.1节) 无法提供强的异常时安全保证。如果在

复制期间抛出异常，它不能保证将自己的目标留在未变动的状态。这样，如果在`copy()`期间`T::operator=()`抛出异常，就不能保证被赋值的`vector`等于给它赋值的那个`vector`，也不能保证该`vector`没有改变。例如，可能其前5个元素已经变成赋值向量相应元素的副本，而其他的没有改变。也有可能出现这种情况，某个元素（那个当`T::operator=()`抛出异常时正在复制中的元素）最后的值既不是该元素原来的值，也不是赋值向量中对应元素的副本。当然，只要`T::operator=()`保证在抛出异常时能将其操作对象留在一个合法状态，整个`vector`也仍然处于一个合法状态——即使它没有处在我们所希望的某个状态之中。

在这里，我还用了一个赋值语句去复制分配器。这实际上并不必要，因为每个分配器都支持赋值（19.4.3节），另见E.8[9]。

标准库`vector`的赋值提供了像最后这个实现一样的较弱的异常时安全性质，以及它潜在的性能优势。也就是说，`vector`的赋值提供的是基本保证，所以它符合大部分人有关异常时安全性的认识。无论如何，它没有提供强保证（E.2节），如果你需要有一个赋值操作，要求在抛出异常时它能保证`vector`不会改变，你就必须去采用另一个提供了强安全性的实现，或者是自己提供一个赋值操作。例如，

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // “明显的” a = b
{
    vector<T,A> temp(a.get_allocator());
    temp.reserve(b.size());
    for (typename vector<T,A>::iterator p = b.begin(); p!=b.end(); ++p)
        temp.push_back(*p);
    swap(a,temp);
}
```

如果没有足够的存储供`temp`使用去创建`b.size()`个元素的空间了，`std::bad_alloc`就会被抛出，这时并没有对`a`做任何改变。与此类似，如果`push_back()`因为任何原因失败，`a`也将保持不变，因为我们是使用`temp`使用`push_back()`，而不是对`a`。在这种情况下，所有由`push_back()`创建的`temp`元素都将在导致失败的异常重新被抛出之前销毁。

交换并不复制`vector`的元素，它只是简单地交换`vector`的数据成员，也就是说，它交换`vector_base`。因此，即使对元素的操作可能抛出异常，这个交换操作也不会抛出（E.4.3节）。由此可知，`safe_assign()`不会做多余的元素复制，也是相当有效的。

与常见情况一样，也存在代替这种明显实现的其他方式。我们可以让标准库为我们执行将元素复制到`temp`里的工作：

```
template<class T, class A>
void safe_assign(vector<T,A>& a, const vector<T,A>& b) // 简单的a = b
{
    vector<T,A> temp(b); // 将b的元素复制到临时变量
    swap(a,temp);
}
```

实际上，我们可以直接采用按值调用（7.2节）：

```
template<class T, class A>
void safe_assign(vector<T,A>& a, vector<T,A> b) // 简单的a = b（注意，b是值传递的参数）
{
    swap(a,b);
}
```

`safe_assign()` 的最后两个版本都没有复制 `vector` 的分配器，这也是一种允许的优化；参见 19.4.3 节。

E.3.4 `push_back()`

从异常时安全性的观点看，`push_back()` 很像是赋值，在这里我们也必须关注在加入新元素失败时，保证 `vector` 没有改变的问题：

```
template< class T, class A>
void vector<T,A>::push_back(const T& x)
{
    if (space == last) { // 没有可用空间了；重新分配：
        vector_base b(alloc, size() ? 2 * size() : 2); // 加倍分配空间
        uninitialized_copy(v, space, b.v);
        new(b.space) T(x); // 将x的一个副本放到*b.space (10.4.11节)
        ++b.space;
        destroy_elements();
        swap<vector_base<T,A>>(b, *this); // 交换表示
        return;
    }
    new(space) T(x); // 将x的一个副本放到*space (10.4.11节)
    ++space;
}
```

很自然，用于初始化 `*space` 的复制构造函数可能会抛出异常。如果发生这种事件，`vector` 的值将保持不变，其 `space` 也保持不增长。在这种情况下，`vector` 的元素并没有重新分配，所以指向它们的迭代器也不会变得非法。这样，这种实现方式就提供了强保证：当分配器抛出的异常甚至用户提供的复制构造函数抛出的异常时，都能保持 `vector` 不变化。标准库对 `push_back()` 提供了这种保证 (E.4.1 节)。

请注意，在这里没有 `try` 块（除了隐藏在 `uninitialized_copy()` 里的一个）。这里的更新是通过仔细安排顺序的操作完成的，所以，在有异常抛出时 `vector` 将保持不变。

通过安排操作顺序和“资源申请既初始化”技术 (14.4 节) 获得异常时安全性是一种更好的途径，这样做比显式地通过 `try` 块处理错误更优美也更有效。与缺乏特殊异常处理代码相比，程序员以很糟糕的顺序排列代码可能会造成更多与异常时安全性有关的问题。排列顺序的基本规则就是：在替代物被构造好且已经能进行赋值且不可能再抛出异常之前，不要去破坏原有的信息。

异常的出现会导致未预料到的控制流，从而可能产生某些令人诧异的情况。对于一段只有简单局部控制流的代码示例，例如只有 `operator=()`、`safe_assign()` 和 `push_back()` 等，产生令人诧异情况的机会很有限。查看这种代码并自问“这行代码会抛出异常吗？如果抛出它会出现什么情况？”这时事情做起来相对比较简单。而对那些大的函数，它们具有复杂的控制结构，如复杂的条件语句和嵌套的循环，回答这种问题就困难多了。加入 `try` 块将增加其中局部控制结构的复杂性，因此也会成为导致混乱和错误的根源 (14.4 节)。我想，安排操作顺序的方法和“资源申请既初始化”方法比广泛使用 `try` 块更有效，其根源就在于对局部控制流的简化。简单的规范化的代码更容易理解，因此也更容易做正确。

注意，这里介绍 `vector` 只是作为一个例子，以便讨论异常可能造成的问题和直面这些问题的各种技术。标准并没有要求实现必须做得像这里所介绍的一样。标准做出了哪些保证是 E.4 节的主题。

E.3.5 构造函数和不变式

从异常时安全性的角度看，其他`vector`操作或者与前面已经考察过的操作等价（因为它们也以类似的方式申请和释放资源），或者完全是平凡的（因为它们根本就不执行需要巧妙地去维护合法状态的操作）。然而，对于大部分类而言，这种“平凡”函数组成了代码中主要的部分。写出这些代码的困难程度与构造函数创建起来供它们在其中操作的那个环境密切相关。换一种说法，“常规成员函数”的复杂性与好的类不变式（24.3.7.1节）的选择密切相关。通过检验那些“平凡的”`vector`函数，我们有可能取得一些对如下有趣问题的见识：怎样才算是一个好的类不变式，应该如何写构造函数去建立起这样的不变式。

像`vector`的下标（16.3.3节）这样的操作很容易写，因为它们可以依靠由构造函数建立起的、由所有申请和释放资源的函数维护着的不变式。特别是，下标操作可以依靠`v`引用着一个元素数组的事实：

```
template< class T, class A>
T& vector<T,A>::operator[] (size_type i)
{
    return v[i];
}
```

让构造函数去申请资源并建立起一个简单的不变式，这是最重要最根本性的要求。为了看清个中缘由，现在考虑`vector_base`的另一种定义：

```
template<class T, class A = allocator<T> > // 构造函数的笨拙使用
class vector_base {
public:
    A alloc; // 分配器
    T* v; // 所分配空间的开始
    T* space; // 元素序列结束，已分配供可能扩充的空间开始
    T* last; // 所分配空间的结束

    vector_base(const A& a, typename A::size_type n) : alloc(a), v(0), space(0), last(0)
    {
        v = alloc.allocate(n);
        space = last = v+n;
    }

    ~vector_base() { if (v) alloc.deallocate(v, last-v); }
};
```

在这里，我分两阶段构造出一个`vector_base`：首先建立起一个安全状态，其中`v`、`space`和`last`都设置为0；在做完这些之后我才去试着分配存储。这种做法是基于一种错位的担心：害怕在元素分配期间发生异常，可能留下一个部分构造的对象。说这种担心是错位的，是因为不可能出现“留下”部分构造对象而后被访问的情况。有关静态对象、自动对象、成员对象和标准库容器的规则都不允许这种情况的发生。然而，这种情况过去/现在可能在标准之前的库中发生，那里过去/现在使用的是放置式的`new`（10.4.11节）去构造容器中的对象，设计中也并没有考虑异常时安全性。打破老习惯确实很困难。

注意，企图这样写出更安全的代码也会使类的不变式复杂化：这个类不再能保证`v`一定指向分配好的存储，现在`v`也可能是0，这是立即出现的代价。标准库对分配器的要求并不保证我们能安全地去释放值为0的指针（19.4.1节），这是分配器与`delete`不同的地方（6.2.6节）。因此我必须在析构函数里写一个检测。还有，每个元素都是先初始化而后再赋值，对于那些赋

值操作并不平凡的元素类型（如`string`和`list`），完成这种额外工作的代价就可能变得很大。

这种两阶段构造并不是一种罕见的风格。有时还被人明确化，让构造函数只做一些“简单而安全的”初始化工作，将对象置入一个可析构的状态，把真正的构建工作留给一个`init()`函数完成，并要求用户必须显式地去调用它。例如，

```
template<class T>    // 古旧（标准之前，有异常之前）风格
class vector_base {
public:
    T* v;           // 所分配空间的开始
    T* space;       // 元素序列结束，已分配供可能扩充的空间开始
    T* last;        // 所分配空间的结束

    vector_base() : v(0), space(0), last(0) {}
    ~vector_base() { free(v); }

    bool init(size_t n) // 如果初始化成功就返回true
    {
        if (v = (T*) malloc(sizeof(T)*n)) {
            uninitialized_fill(v, v+n, T());
            space = last = v+n;
            return true;
        }
        return false;
    }
};
```

这一风格所声称的价值是：

- [1] 构造函数不会抛出异常，采用`init()`做初始化的成功与否可以通过“普通的”（即非异常的）方式测试。
- [2] 存在很简单的合法状态，在遇到严重问题时，操作可以让对象处于这种状态。
- [3] 对资源的请求可以推迟到实际需要完全初始化的对象时再做。

下面几小节将考察这些论点，并说明为什么这种两阶段构造技术不能带来它所期望的利益。它还会变成一些问题的根源。

E.3.5.1 使用`init()`函数

第一点（用一个`init()`函数比用构造函数更好）是骗人的。采用构造函数和异常处理，是对付资源申请和初始化错误的一种更通用、更系统化的方式（14.1节、14.4节）。这里所采用的风格是还没有异常机制之前的C++的遗风。

采用两种风格细心写出的代码大致等价。请考虑

```
int f1(int n)
{
    vector<X> v;
    // ...
    if (v.init(n)) {
        // 将v用做n个元素的vector
    }
    else {
        // 处理问题
    }
}
```

和


```
int f2(int n)
try {
    vector v<X> v(n);
    // ...
    // 将v用做n个元素的vector
}
catch (...) {
    // 处理问题
}
```

然而，采用一个独立的`init()`函数就给下面问题创造了机会：

- [1] 忘记调用`init()` (10.2.3节)。
- [2] 忘记测试`init()`的成功与否。
- [3] 忘记`init()`可能抛出一个异常。
- [4] 在调用`init()`之前使用对象。

`vector<T>::init()`的定义就说明了第[3]点。

在一个很好的C++实现里，`f2()`还将略微比`f1()`快一点，因为它在最普遍的情况下避免了测试。

E.3.5.2 依赖于某个默认的合法状态

第二点（有一个容易构造出的“默认”合法状态）一般说是正确的，但是对于`vector`的情况，为得到这一点付出了不必要的代价。现在就可能出现`vector_base`中`v == 0`的情况，所以在`vector`实现里就必须自始至终防备这种可能性。例如，

```
template< class T>
T& vector<T>::operator[] (size_t i)
{
    if (v) return v[i];
    // 处理错误
}
```

由于打开了`v == 0`的可能性，就使无范围检查的下标操作等价于有范围检查的访问了：

```
template< class T>
T& vector<T>::at(size_t i)
{
    if (i < v.size()) return v[i];
    throw out_of_range("vector index");
}
```

在这里发生的最本质情况是，由于引进了`v == 0`的可能性，我使`vector_base`的基本不变式变得更复杂了。由此，`vector`的基本不变式也类似地变复杂了。这一情况的最终结果就是`vector`和`vector_base`里的所有代码处理起来都更复杂了。这些将成为潜在的错误、维护中的问题、运行时的额外开销的根源。注意，条件语句在现代机器结构中的代价可能令人感到吃惊。在所有需要考虑效率的地方，在实现关键性的操作如向量下标操作时，避免条件语句都是至关重要的事情。

有意思的是，`vector_base`的原来定义中已经存在着一个很容易构造的合法状态。除非初始分配成功，否则根本就不会有`vector_base`对象存在。因此，`vector`的实现者可以按如下方式写出一个“紧急出口”函数：

```
template< class T, class A>
void vector<T, A>::emergency_exit()
```

```
{
    space = v;           // 将 *this 的大小设置为0
    throw Total_failure();
}
```

这样做过于急躁了一点，因为它没有去调用元素的析构函数，没有去释放由 `vector_base` 保存的这些元素而用的空间，也就是说，它没有提供基本保证（E.2节）。如果我们欣然地信任 `v` 和 `space` 的值和元素的析构函数，我们就能避免潜在的资源流失：

```
template< class T, class A>
T& vector<T,A>::emergency_exit()
{
    destroy_elements(); // 清理
    throw Total_failure();
}
```

请注意，标准 `vector` 具有这样一种清晰的设计，它尽可能地避免了两阶段构造所产生的问题。`init()` 函数大致上相当于 `resize()`，在大部分情况中 `v == 0` 的可能性已经被 `size() == 0` 所覆盖了。当我们考虑申请重要资源的应用类（例如网络连接和文件）时，这里所说的两阶段构造的负面影响将变得更加明显。这种类很少成为某种指导着它们的设计和使用的框架的一部分，不会像标准库的需求指导着 `vector` 的定义和使用那样。当应用中的概念与为实现它们所需的资源之间的关系变得更复杂时，这些问题就会进一步复杂化。很少有类能像 `vector` 这样与系统资源之间有着如此直接的映射。

有一个“安全状态”在原则上是一个很好的想法。但是，如果我们不担忧在完成操作之前会抛出异常，就无法将对象置入一个合法状态，我们一定是有了一个大问题。无论如何，这种“安全状态”应该是为类的自然语义的一部分的状态，而不是某种完全为了实现的、而且使类的不变式复杂化的人为现象。

E.3.5.3 延迟资源申请

与第二点（E.3.5.2节）类似，第三点（将资源申请推迟到需要时再做）也是把一个好想法用错了地方，增加了代价而又没有产生效益。在许多情况下，特别是对 `vector` 这样的容器，延迟资源申请的最佳方式就是让程序员将创建对象推迟到需要它们的时候。考虑 `vector` 的朴素使用：

```
void f(int n)
{
    vector<X> v(n); // 做出n个类型X的默认对象
    // ...
    v[3] = X(99); // 真正的v[3]“初始化”
    // ...
}
```

构造出一个 `X` 仅仅是为了后来给它赋一个新值是一种浪费，特别是当 `X` 的赋值代价高昂时。由此看来，两阶段创建 `X` 看起来好像很有吸引力。例如，类型 `X` 本身可能就是 `vector`，因此我们可以考虑 `vector` 的两阶段构造，设法去优化空 `vector` 的创建。然而，创建默认的（空）向量已经效率很高了，因此为空向量的特殊情况而使实现复杂化看来是毫无益处的。更一般地看，解决这种荒谬的初始化，最好的办法通常不是从元素的构造函数里去掉复杂的初始化，与此相反，用户应该在需要时再去创建元素。例如，

```

void f2(int n)
{
    vector<X> v;           // 做出一个空向量
    // ...

    v.push_back(X(99));    // 左需要时创建元素
    // ...
}

```

总结一下，两阶段构造的方法将导致更复杂的不变式，通常也将导致更不优美、更容易出错、更难维护的代码。因此，只要可行，就应该优先选用语言所支持的“构造函数方法”，而不是“*init()* 函数方法”。也就是说，只要延迟资源申请并不是某个类的内在语义要求，就应该在构造函数里申请资源。

E.4 标准容器的保证

如果一个库操作本身抛出了异常，它应该能而且确实也做出一种保证，保证它正在操作的那个对象将处于一种定义良好的状态。例如，*at()* 对某个 *vector* 抛出 *out_of_range* 异常（16.3.3节），不会对 *vector* 的异常时安全性造成问题。写 *at()* 的人可以无疑地确信，在异常抛出前 *vector* 处于一个定义良好的状态。对于库的实现者、库的用户和试图理解代码的人而言，问题在用户所提供的函数抛出异常时出现。

标准库容器提供了基本保证（E.2节）：只要用户的代码具有所需要的行为方式，库的基本不变式就能够维持，而且不会有资源流失。也就是说，用户提供的操作不应该将容器元素留在非法状态里，不应从析构函数里抛出异常。这里的“操作”，我说的是标准库实现要使用的那些操作，如构造函数、赋值、析构函数和在迭代器上的操作等（E.4.4节）。

相对而言，程序员保证这些操作满足标准库的要求是比较容易的事情。事实上，许多朴素写出的代码都能符合库的要求。下面几种类型显然都满足标准库对容器元素类型的要求：

- [1] 内部类型，包括指针。
- [2] 没有用户定义操作的类型。
- [3] 其操作不抛出异常也不会将操作对象留在非法状态的类。
- [4] 具有下列特征的类：其析构函数绝不抛出异常，而且很容易验证标准库所使用的那些操作（例如构造函数、赋值、*<*、*==* 和 *swap()*）都不会将操作对象留在非法状态。

对于每种情况，我们还必须保证不出现存储流失。例如，

```

void f(Circle* pc, Triangle* pt, vector<Shape*>& v2)
{
    vector<Shape*> v(10);           // 或创建vector或抛出bad_alloc
    v[3] = pc;                      // 不会抛出异常
    v.insert(v.begin()+4, pt);      // 或插入pt，或对v无影响
    v2.erase(v2.begin()+3);         // 或删除v2[3]，或对v2无影响
    v2 = v;                         // 或复制v，或对v2无影响
    // ...
}

```

在 *f()* 退出时，*v* 将被正确销毁，而且 *v2* 将处于一种合法状态。这段代码并没有说谁负责删除 *pc* 和 *pt*。如果这是 *f()* 的责任，它就可以捕捉异常而后完成必要的删除，或者将这些指针赋值给局部的 *auto_ptr*。

更有趣的问题是：什么时候库操作能够提供一个操作或者成功或者对其操作对象毫无影

响的强保证？例如，

```
void f(vector<X>& vx)
{
    vx.insert(vx.begin()+4, X(7));    // 加入元素
}
```

一般说， X 的操作和 $\text{vector}<X>$ 的分配器可能抛出异常。当 $f()$ 由于一个异常而退出时，我们能对 vx 的元素说些什么呢？基本保证确定不会有资源流失，而且 vx 将保持着一些组合的元素。但是，究竟是什么元素？ vx 改变了吗？可能加入一个默认的 X 吗？可能删除了某个元素，如果是使 $\text{insert}()$ 能在维持基本保证时恢复出来的惟一方式？有时知道一个容器处在某个良好的状态还不够，我们还希望知道究竟是在什么状态。在捕捉到一个异常之后，我们常希望知道所有元素正是我们所需要的，或者我们应该开始做错误恢复操作。

E.4.1 元素的插入和删除

在容器中插入或者删除一个元素是操作的明显例子，如果抛出了异常，这些操作就可能将容器留在某个未曾预料的状态。因为插入和删除都需要调用许多可能抛出异常的操作：

- [1] 向容器中复制一个新值。
- [2] 从容器里删除的一个元素必须销毁。
- [3] 有时必须做存储分配以保存新元素。
- [4] 有时必须复制 vector 和 deque 的所有元素以取得新的空位。
- [5] 关联容器需要对元素调用比较函数。
- [6] 许多插入和删除涉及到迭代器操作。

这些情况中的每一个都可能导致抛出异常。

如果析构函数抛出了异常，那么就不可能做出任何保证了（E.2节）。要在这种情况下做出任何保证，都将付出无法容忍的高代价。然而，库有可能也确实在保护它自己及其用户，使之不受由用户提供的其他操作中抛出的异常的伤害。

在操纵一个链接式数据结构（例如 list 或者 map ）时，可以加入或者删除元素而不影响容器中的其他元素。对那些采用连续分配的方式保存元素的容器（例如 vector 或者 deque ），情况就不是这样，在这里有时需要将所有元素移到新的位置。

除了基本保证之外，标准库还为一些插入和删除元素的操作提供了强保证。由于实现为链接数据结构的容器与为元素分配连续位置的容器的行为不同，标准对不同种类的容器所提供的保证也略有差异：

- [1] 对 vector （16.3节）和 deque （17.2.3节）的保证：

- 如果异常由 $\text{push_back}()$ 或 $\text{push_front}()$ 抛出，该函数不会造成任何影响。
- 除非是由元素类型的复制构造函数或者赋值抛出，否则，如果 $\text{insert}()$ 抛出异常，该函数不会造成任何影响。
- 除非元素类型的复制构造函数或者赋值抛出，否则 $\text{erase}()$ 不会抛出异常。
- $\text{pop_back}()$ 或 $\text{pop_front}()$ 不抛出异常。

- [2] 对 list （17.2.2节）的保证：

- 如果异常由 $\text{push_back}()$ 或 $\text{push_front}()$ 抛出，该函数不会造成任何影响。
- 如果 $\text{insert}()$ 抛出异常，该函数不会造成任何影响。

- `erase()`、`pop_back()`、`pop_front()`、`splice()` 或 `reverse()` 不会抛出异常。
- 除非所用谓词或者比较函数抛出异常，否则 `list` 的成员函数 `remove()`、`remove_if()`、`unique()`、`sort()` 和 `merge()` 都不会抛出异常。

[3] 对关联容器（17.4节）的保证：

- 如果在插入单个元素时异常由 `insert()` 抛出，该函数没有任何影响。
- `erase()` 不会抛出异常。

注意，在对某种容器的一个操作提供了强保证的地方，如果抛出异常，所有迭代器、指向元素的指针、对元素的引用都仍然是合法的。

这些规则总结在下表中：

容器操作的保证				
	vector	deque	list	map
<code>clear()</code>	nothrow (复制)	nothrow (复制)	nothrow	nothrow
<code>erase()</code>	nothrow (复制)	nothrow (复制)	nothrow	nothrow
<i>I</i> -元素 <code>insert()</code>	strong (复制)	strong (复制)	strong	strong
<i>N</i> -元素 <code>insert()</code>	strong (复制)	strong (复制)	strong	basic
<code>merge()</code>	—	—	nothrow (比较)	—
<code>push_back()</code>	strong	strong	strong	—
<code>push_front()</code>	—	strong	strong	—
<code>pop_back()</code>	nothrow	nothrow	nothrow	—
<code>pop_front()</code>	—	nothrow	nothrow	—
<code>remove()</code>	—	—	nothrow (比较)	—
<code>remove_if()</code>	—	—	nothrow (谓词)	—
<code>reverse()</code>	—	—	nothrow	—
<code>splice()</code>	—	—	nothrow	—
<code>swap()</code>	nothrow	nothrow	nothrow	nothrow (比较的复制)
<code>unique()</code>	—	—	nothrow (比较)	—

在这个表里：

- basic** 表示操作只提供基本保证（E.2节）。
- strong** 表示操作提供强保证（E.2节）。
- nothrow** 表示操作不抛出异常（E.2节）。

— 表示这个容器没有提供此操作作为成员。

当一种保证要求用户提供的某些操作不抛出异常时，有关的操作被写在保证类别之下的括号里面。这些要求都已经在表前的正文中精确地说明过了。

在这里，*swap()* 函数与所提到的其他函数的差异在于它们并不是成员。对 *clear()* 的保证是由 *erase()* 的保证推导出来的（16.3.6节）。这个表列出了基本保证之上的有关事项。因此，未列入这个表里的那些操作，例如 *vector* 的 *reverse()* 和 *unique()*，就都是只提供了对所有序列的算法，而没有提供进一步的保证。

“拟容器”的 *basic_string*（17.5节、20.3节）对所有操作提供了基本保证（E.5.1节）。标准库还保证 *basic_string* 的 *erase()* 和 *swap()* 不会抛出异常，并为 *basic_string* 的 *insert()* 和 *push_back()* 提供了强保证。

如果一个操作提供了强保证，它除了保证容器没有改变之外，还保证所有的迭代器、指针和引用都仍然合法。例如，

```
void update (map<string, X>& m, map<string, X>::iterator current)
{
    X x;
    string s;
    while (cin >> s >> x)
        try {
            current = m.insert(current, make_pair(s, x));
        }
        catch (...) {
            // 在这里current仍指称着当前元素
        }
}
```

E.4.2 保证和权衡

这种有关附加保证的拼凑性工作也反应了实现中的现实情况。程序员希望得到带有尽可能少的条件的强保证，但他们也倾向于强调每个个别标准库操作都应该有最佳效率。这两种愿望都非常合理，但是，对于大多数操作而言却不可兼得。为了使入对所涉及的权衡情况有一个更好的看法，我将考察向 *list*、*vector* 和 *map* 中加入一个和多个元素的各种方式。

考虑将一个元素加入某 *list* 或者 *vector*。如常，*push_back()* 是为完成此事面提供的最简单的方式：

```
void f(list<X>& lst, vector<X>& vec, const X& x)
{
    try {
        lst.push_back(x);          // 加入list
    }
    catch (...) {
        // lst未改变
        return;
    }
    try {
        vec.push_back(x);          // 加入vector
    }
    catch (...) {
        // vec未改变
        return;
    }
}
```

```

    }
    // lst和vec各有一个值为x的新元素
}

```

在这些情形中，提供强保证既简单又廉价。这也非常有用，因为它提供的是一种具有完全的异常时安全性的加入元素方式。然而，没有为关联容器定义`push_back()`——对于`map`无`back()`可言。归根结底，关联容器的最后元素是由顺序关系定义的，而不是通过位置。

对`insert()`的保证就有点复杂了，原因在于`insert()`有时必须把元素放到容器的“中间”。对链接式结构，完成此事不会有任何问题。另一方面，如果在一个`vector`里还存在空闲的保留空间，`vector<X>::insert()`的最明显实现方式就是复制插入点之后的元素以腾出空位。这样做在效率上是最优的，但如果`X`的复制赋值或者复制构造函数抛出异常，我们就没有简单的方法去恢复`vector`了（见E.8[10~11]）。因此，`vector`所提供的保证是有条件的，基于元素复制操作不抛出异常。而`list`和`map`就不需要这种条件，它们可以在完成所有复制工作之后，简单地将元素链入。

作为例子，假设`X`的复制赋值和复制构造函数在无法成功创建副本时抛出`X::cannot_copy`：

```

void f(list<X>& lst, vector<X>& vec, map<string,X>& m, const X& x, const string& s)
{
    try {
        lst.insert(lst.begin(), x);    // 加入list
    }
    catch (...) {
        // lst未改变
        return;
    }

    try {
        vec.insert(vec.begin(), x);    // 加入vector
    }
    catch (X::cannot_copy) {
        // 呜呼：vec可能有也可能没有新元素
        return;
    }
    catch (...) {
        // vec未改变
        return;
    }

    try {
        m.insert(make_pair(s, x));    // 加入map
    }
    catch (...) {
        // m未改变
        return;
    }

    // lst和vec都有一个值为x的新元素
    // map有一个值为(s, x)的新元素
}

```

如果捕捉到`X::cannot_copy`，那么新元素可能已经插入`vec`，也可能尚未插入。如果新元素已经插入，它一定是一个处于合法状态的元素，但到底是什么值则没有明确的规定。完全可能在`X::cannot_copy`之后，有些元素将“神秘地”出现了重复（E.8[11]）。也可能以另一种方式实

现`insert()`，使得它可能删除掉某些“尾部”元素，以确保在容器里不会留下非法的元素。

不幸的是，不提出复制操作中不得抛出异常的要求，要为`vector`的`insert()`提供强保证实际上不可行。与在移动`vector`元素的工作中提供基本保证相比，在同样情况下提供完全的保护，以防异常的代价可能太大了。

常常能看到有些元素类型的复制操作存在抛出异常的可能性。来自标准库本身的例子如`vector<string>`、`vector<vector<double>>`和`map<string, int>`等。

`list`和`vector`容器为一个元素和多个元素的`insert()`提供了同样保证。原因很简单，对于`list`和`vector`，单个和多个元素的`insert()`采用了同样的实现策略。而`map`为单个元素的`insert()`提供了强保证，但为多元素的`insert()`只提供了基本保证。对于`map`，提供强保证的单元素`insert()`很容易实现。然而，为`map`实现多元素插入的明显策略就是一个接一个地插入，为这种方式提供强保证就不容易了。这里的问题是，在插入某个元素失败时，不存在一种简单方式退掉前面已成功完成的插入操作。

如果真的希望一个插入函数能提供强保证：或者是所有元素都成功插入，或者操作不造成任何影响，我们也是可以做到的，只要先构造起一个容器而后再`swap()`：

```
template<class C, class Iter>
void safe_insert(C& c, typename C::const_iterator i, Iter begin, Iter end)
{
    C tmp(c.begin(), i);           // 将前部元素复制到临时量
    copy(begin, end, inserter(tmp, tmp.end())); // 复制新元素
    copy(i, c.end(), inserter(tmp, tmp.end())); // 复制后续元素
    swap(c, tmp);
}
```

与平常一样，如果元素的析构函数抛出异常，这段代码也会有错误的行为。然而，如果某个元素复制操作抛出了异常，参数容器就会保持不变。

E.4.3 交换

与复制构造函数和赋值类似，`swap()`操作也是许多标准算法中的最基本操作，常常由用户提供。例如，`sort()`和`stable_sort()`通常都用`swap()`重新排序元素。这样，如果一个`swap()`函数在交换来自容器的值期间抛出了异常，该容器有可能留在改变之前的状态，也有可能出现重复元素而不是交换过的元素。

考虑标准库`swap()`函数（18.6.8节）的明显定义方式：

```
template<class T> void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

很清楚，`swap()`不会抛出异常，除非元素类型的复制构造函数或者复制赋值抛出异常。

除了关联容器方面的一个次要的例外情况，标准容器的`swap()`函数都不会抛出异常。简而言之，容器的交换都是通过交换数据结构完成的，在功能上，这种数据结构就像是到元素的句柄（13.5节、17.1.3节）。因为元素本身不移动，因此就不会调用元素的构造函数或者赋值，它们也就没有机会抛出异常了。此外，标准库的`swap()`函数还能保证引用到被交换容器

中元素的任何引用、指针或迭代器都不会变为非法。这样，为异常留下的就只有一个潜在的根源：关联容器的比较对象将作为句柄的一部分同时进行复制。在标准容器的`swap()`里，仅有的可能异常就是容器中比较对象的复制构造函数和赋值操作（17.1.4.1节）。幸运的是，比较对象通常都采用极平凡的复制操作，其中不可能抛出异常。

用户提供的`swap()`也应该写得能提供同样的保证。做到此事相对而言比较简单，人们只要记住，要交换表示为句柄的类型只需要交换其句柄，不必去缓慢而煞费苦心地复制由句柄引用的所有信息（13.5节、16.3.9节、17.1.3节）。

E.4.4 初始化和迭代器

分配元素所用存储和对这些存储的初始化是每个容器实现中的主要部分（E.3节）。因此，在未初始化存储中的构造对象的标准算法（`uninitialized_fill()`、`uninitialized_fill_n()`和`uninitialized_copy()`（19.4.4节））都保证：如果它们抛出异常，就不会留下已构造的对象。它们提供的是强保证（E.2节）。这其中有时涉及到销毁元素，因此，对于这些算法而言，析构函数不抛出异常就是最基本的要求了；见E.8[14]。此外，也要求作为参数提供给这些算法的迭代器具有良好的行为，也就是说，它们必须是合法的迭代器，引用合法的序列，而且在合法迭代器上的迭代器操作（如`++`、`!=`和`*`）不允许抛出异常。

迭代器是标准算法和对标准容器的操作中允许自由复制的对象的例子。这样，迭代器的复制构造函数和复制赋值操作都不应该抛出异常。特别是，标准保证由标准容器返回的迭代器的复制构造函数和复制赋值操作都不抛出异常。例如，可以复制由`vector<T>::begin()`返回的迭代器，而根本不必担心异常问题。

注意，迭代器的`++`和`--`操作也有可能抛出异常。例如，`istreambuf_iterator`（19.2.6节）就可能很合理地抛出一个异常，指明出现了输入错误；一个检查访问范围的迭代器可能抛出异常，指明企图到它的合法范围之外去访问（19.3节）。然而，在将迭代器从序列中一个元素移向另一个元素而又不违反在迭代器上的`++`和`--`的定义时，它们是不会抛出异常的。`uninitialized_fill()`、`uninitialized_fill_n()`和`uninitialized_copy()`假定，对于它们的迭代器参数的`++`和`--`都不会抛出异常；如果这些操作真的抛出了异常，那么，这些“迭代器”或者不是符合标准的迭代器，或者由它们所描述的“序列”并不是一个序列。再说一次，标准容器不可能在遇到用户本身的无定义行为时为用户提供保护（E.2节）。

E.4.5 对元素的引用

在将某容器的一个引用、指针或迭代器递交给某段代码时，该段代码可以通过破坏对应的元素去破坏这个容器。例如，

```
void f(const X& x)
{
    list<X> lst;
    lst.push_back(x);
    list<X>::iterator i = lst.begin();
    *i = x;    // 复制x到表里
    // ...
}
```

如果`x`被破坏，`lst`的析构函数可能就无法正确销毁`lst`了。例如，

```

struct X {
    int* p;

    X() { p = new int; }
    ~X() { delete p; }
    // ...
};

void malicious()
{
    X x;
    x.p = reinterpret_cast<int*>(7);    // 破坏x
    f(x);                               // 定时炸弹
}

```

在执行到`f()`的结束处时将调用`list<X>`的析构函数，该函数将转而去对已经破坏的值调用`X`的析构函数。当`p`不是`0`又不指向一个`X`时，执行`delete p`的效果无定义，而且很可能是立即崩溃。另一种情况，它也完全可能以某种方式去破坏自由存储，从而到很晚以后，在程序的另一个看起来完全无关的部分造成一个难以跟踪的问题。

存在这种破坏的可能性，但也不应该据此去阻止人们通过引用或迭代器来操作容器的元素，因为这通常是完成工作的最简单最高效的方式。当然，明智的做法是特别当心这类索引到容器里的引用。在容器的完整性具有特别重要意义的地方，可能值得为比较缺乏经验的用户提供其他更安全的方式。例如，我们可以提供一个操作，该操作在向重要的容器内复制新元素之前，先对它的值做合法性检查。当然，必须在有了对于应用领域中类型的知识之后，才有可能去做这种检查。

一般来说，如果容器里的一个元素遭到破坏，随后对元素的操作就可能以某种极其险恶的方式失败。这并不是容器的专利，任何处于非法状态的对象都可能导致后续的灾难。

E.4.6 谓词

大部分标准算法和标准容器的许多操作都依赖于一些可以由用户提供的谓词。特别是所有关联容器都要依靠谓词去查找和插入。

标准容器操作中所用的谓词有可能抛出异常。在这种情况下，所有标准库操作都提供了基本保证；而某些操作（如对单个元素的`insert()`）提供了强保证（E.4.1节）。如果某个谓词从一个容器的操作中抛出异常，留在这个容器里的元素集合未必恰好是用户所需要的，但它仍然是一组合法的元素。例如，假如在调用`list::unique()`（17.2.2.3节）时`==`抛出了异常，用户就不能假定表里没有重复元素。用户能够假定的所有东西就是：表中的每一个元素都是合法的（见E.5.3节）。

幸好谓词极少做某些可能抛出异常的事情。当然，在考虑异常时安全性问题时，也必须将用户定义的`<`、`==`和`!=`谓词考虑在内。

关联容器的比较对象将由`swap()`复制，作为交换的一部分（E.4.3节），因此，保证那些可能被用做比较对象的谓词的复制操作不抛出异常也是一个好主意。

E.5 标准库的其他方面

在异常时安全性考虑中，最关键的问题是维护对象的一致性；也就是说，我们必须维护各个对象的基本不变式，还要维护对象集合的一致性。在标准库的环境里，最难提供异常时

安全性的对象就是那些容器。从异常时安全性的角度看，标准库的其他部分并不那么令人感兴趣。当然，从异常时安全性的角度可以看到，内部数组是一种经过一次不安全操作就可能破坏的容器。

一般来说，标准库函数只抛出它们描述了的那些异常，再加上它们可能调用的用户提供操作所抛出的异常。此外，任何（直接或者间接）分配存储的函数都可能抛出表明存储耗尽的异常（典型情况中是`std::bad_alloc`）。

E.5.1 串

对`string`的操作可能抛出许多不同的异常。然而，`basic_string`通过由`char_traits`（20.2节）所提供的函数操纵自己的字符，而且这些函数都不允许抛出异常。也就是说，由标准库支持的`char_traits`都不抛出异常，如果某个用户定义`char_traits`中的操作抛出了异常，那就没有任何保证了。特别值得注意的是用做`basic_string`的元素（字符）类型的类型不允许有用户定义的复制构造函数或者用户定义的复制赋值。这也就消除了抛出异常的一个重要根源。

`basic_string`很像一种标准容器（17.5节、20.3节）。事实上，它的元素也组成了一个可以通过`basic_string<Ch, Tr, A>::iterator`和`basic_string<Ch, Tr, A>::const_iterator`访问的序列。因此，一个串的实现提供了基本保证（E.2节），对`erase()`、`insert()`、`push_back()`和`swap()`（E.4.1节）的保证都适用于`basic_string`。例如，`basic_string<Ch, Tr, A>::push_back()`也提供了强保证。

E.5.2 流

如果要求它们那样去做，`iostream`的函数就会通过抛出异常的方式发出状态变化信号（21.3.6节）。这一动作的语义具有良好的定义，而且不会给异常时安全性带来问题。如果某个用户定义的`operator<<()`或者`operator>>()`抛出了异常，它也将被呈现给用户，就像是`iostream`库抛出一个异常似的。不过这种异常并不影响流的状态（21.3.3节）。对流的进一步操作有可能无法找到所期望的数据（因为前面的操作抛出了异常，而不是正常完成了动作），但流本身不会破坏。就像遇到其他I/O问题一样，在进行随后的读/写之前可能需要做一次`clear()`（21.3.3节、21.3.5节）。

与`basic_string`类似，`iostream`依靠`char_traits`去操纵字符（20.2.1节、E.5.1节）。这样，具体实现就可以假设对字符的操作都不会抛出异常；如果用户违反了这个假设，那就没有任何保证了。

为了能做某些至关重要的优化，`locale`（D.2节）和`facet`（D.3节）都被假定不会抛出异常。如果它们真的抛出，使用它们的流就可能被破坏。然而，最可能的异常是来自`use_facet`（D.3.1节）的`std::bad_cast`，这只能出现在标准流实现之外的用户提供的代码中。在最坏情况下，这种情况可能产生出不完整的输出或者导致读入失败，但不会破坏`ostream`（或`istream`）本身。

E.5.3 算法

除了`uninitialized_copy()`、`uninitialized_fill()`和`uninitialized_fill_n()`（E.4.4节）之外，标准只对算法提供了基本保证（E.2节）。也就是说，只要用户提供的对象具有良好行为，这

些算法将能维持所有的标准库不变式，且不会造成资源流失。为避免无定义行为，用户提供的操作应该总将其操作对象留在合法状态，它们的析构函数也不应该抛出异常。

算法本身都不抛出异常。相反，它们都通过它们的返回值报告错误和失败。举例来说，检索算法一般采用返回被检索序列结束的方式说明“没有找到”（18.2节）。这样，从标准算法抛出的异常，其根源必定在于某个用户提供的操作。由此，这种异常必定来自某个对元素的操作（例如一个谓词（18.4节）、一个赋值或者一个`swap()`），或者来自某个分配器（19.4节）。

如果这样一个操作抛出了异常，算法将立即结束，并且将这个异常留给调用算法的那个函数去处理。对于某些算法，异常完全可能在容器正处于某种用户认为很不好的状态之的时刻发生。例如，某些排序算法会临时性地将元素复制到一个缓冲区，而后再将它们复制回容器里。这样的`sort()`就有可能在将元素复制出容器（计划是以后按照正确顺序将它们复制回来），将原来的元素覆盖掉之后抛出了异常。从用户的观点看，这个容器已经被破坏了。然而，因为所有元素都处于合法的状态，恢复它们应该还是比较直截了当的事情。

注意，标准算法都通过迭代器访问序列，也就是说，标准算法绝不直接对容器操作，而只是操作容器的元素。标准算法绝不会对容器直接增加或者减少元素是一个重要事实，这一事实可以简化对于异常所造成影响的分析。与此类似，如果对某个数据结构的访问只是通过到`const`的迭代器、指针或者引用（例如，通过一个`const Rec*`），验证某个异常不会产生人们所不希望的后果通常都很简单。

E.5.4 `valarray`和`complex`

数值函数并不显式地抛出异常（第22章）。不过，`valarray`需要分配存储，这样就可能抛出`std::bad_alloc`。进一步说，有可能给了`valarray`或`complex`一个能抛出异常的元素类型（标量类型）。与其他地方一样，标准库提供了基本保证（E.2节），但是对于由异常结束的计算，其结果就没有任何特别保证了。

与`basic_string`（E.5.1节）类似，`valarray`和`complex`也被允许做出这样的假定：它们的模板参数类型没有用户定义的复制函数，因此可以按照二进制位的方式复制。在典型情况下，这些标准库数值类型都针对速度做了优化，并假定它们的元素类型（标量类型）不会抛出异常。

E.5.5 C标准库

一个不带异常描述部分的标准库操作也可能以某种实现确定的方式抛出异常。然而，来自C标准库的函数都不抛出异常，除非它们函数参数抛出。毕竟这些函数是与C共享的，而C根本就没有异常。一个具体实现也可以给标准C函数的声明加上一个空的异常描述（*exception-specification*）`throw()`来帮助编译器生成更好的代码。

像`qsort()`和`bsearch()`（18.11节）这样的函数以一个函数指针为参数，如果其参数抛出异常，它们也会抛出。基本保证（E.2节）也包括了这些函数。

E.6 对于库用户的寓意

看待标准库环境中的异常时安全性的一种方式就是，我们不会有问题，除非我们自己创造出问题：只要用户提供的操作能符合标准库的基本要求（E.2节），库就能正确工作。特别是，标准容器抛出异常时不会导致存储流失，也不会将容器留在非法状态中。这样，对于库

用户的问题就变成了：我应该如何定义自己的类型，使它们不会导致无定义行为或者造成存储流失呢？

基本规则是：

- [1] 在更新一个对象时，当新的表示完全构造起来，并能在不会出现异常的情况下取代原有表示之前，不要销毁对象原来的表示。举例来说，请看`vector::operator=()`、`safe_assign()`和`vector::push_back()`的实现，在E.3节。
- [2] 在抛出一个异常之前，释放所有过去申请的、现在并没有为某些（其他）对象所拥有的资源。
 - [2a] “资源申请即初始化”技术（14.4节）和语言中有关部分构造的对象将按照它们被构造的程度予以销毁的规则（14.4.1节）在这里有极大的帮助。例如，见E.2节的`leak()`。
 - [2b] 算法`uninitialized_copy()`及其兄弟算法都提供了在完成一组对象的构造工作失败时，自动释放资源的功能（E.4.4节）。
- [3] 在抛出异常之前，确认每个操作对象都处在一个合法状态。也就是说，让每个对象都处在某种状态，使其可以访问和销毁而不会导致无定义行为或从析构函数里抛出异常的情况。举例说，请看E.3.2节的`vector`的赋值。
 - [3a] 注意，构造函数有其特殊性，当构造函数抛出异常时，不会留下需要以后再销毁的对象。这也就意味着，这时我们不必去建立一个不变式，而在抛出异常之前我们必须确认已经释放了在这次失败的构造期间所申请的全部资源。
 - [3b] 注意，析构函数也有特殊性，因为从析构函数抛出的异常几乎可以肯定将导致违反不变式的情况，而且/或者引起调用`terminate()`。

在实践中，按照这些规则行事的困难程度令人吃惊，最根本的原因是：异常可能从人们没有预料到的地方抛出。最好的一个例子就是`std::bad_alloc`。在某些程序里，我们可以通过不用完所有的存储而解决这个特殊问题。然而，对于那些其本意就是要长期运行或需要接受任意数量输入的程序，我们就必须准备好去处理申请资源时的各种故障。这样，我们就必须假定每一个函数都会抛出异常，除非已证明它并非如此。

试着去避免惊诧有一种简单方式，那就是去使用那些元素不会抛出异常的容器（例如，指针的容器或者简单的具体类型的容器），或者采用那些能提供强保证（E.4节）的链接容器（例如`list`）。与之互补的另一种方式是主要依靠某些提供强保证的操作，例如`push_back()`操作，或者完成，或者没有影响（E.2节）。然而，这些操作本身都不足以避免资源流失，而且可能导致一种纯实用的、过于受束缚的、悲观的处理错误和恢复的方法。举例来说，只要对`T`的操作不抛出异常，一个`vector<T*>`就是异常时安全的。但是，除非在某些地方删除被指针所指的對象，否则由这种`vector`抛出的异常就会导致存储流失。因此，引进`Handle`类去处理释放存储的问题（25.7节），并采用`vector<Handle<T>>`而不是普通的`vector<T*>`将能改善代码的可靠性。

在写新代码时，有可能采取一种更系统化的方法，并确保每种资源都由一个类代表，这种类的不变式提供了基本保证（E.2节）。有了这些之后，我们就有可能在应用中标识出最关键的对象，并为针对它们的操作提供一种可以回退的语义（也就是，强保证——可能在某些特殊的条件下）。

大部分应用里都可能包含着一些数据结构和代码，在写这些结构和代码时，人们心里还

没有异常时安全性的概念。如果必要,也可以设法使这些代码适应于某个异常时安全性的框架,方法就是,或者是验证它不会抛出异常(就像C标准库的情况,E.5.5节),或者采用一个异常时的行为和资源管理都能够精确描述的界面类。

在设计一个意欲将其用到某个具有异常时安全性的环境中的类型时,我们必须特别关注标准库将要使用的那些操作:构造函数、析构函数、赋值、比较、交换函数、用做谓词的函数以及针对迭代器的操作。完成这件事的最好方式就是定义一个类不变式,它应该很容易由所有构造函数建立。有时,我们必须设计自己的类不变式,使这种对象可以被放入某种状态,在这种状态中它可以被正确销毁,即使某个操作在一个“不方便的”位置遭遇了失败。理想的情况下,这一状态不应是专为有助于异常处理而定义的一种人为现象,而应是按照类型的语义自然产生的一种状态(E.3.5节)。

在考虑异常时安全性时,应该将重点放在为对象定义合法状态(不变式)和正确释放资源的问题上。将资源直接表示为类也因此显得尤为重要。`vector_base`(E.3.2节)是这方面的一个简单示例。这种类的构造函数申请低级资源(例如`vector_base`申请的原始存储)并建立起不变式(例如`vector_base`里各个指针的正确初始化)。这种类的析构函数隐式地释放低级资源。有关部分构造的规则(14.4.1节)和“资源申请即初始化”技术(14.4节)支撑着这种资源管理方式。

书写良好的构造函数对一个对象建立起类不变式(24.3.7.1节),也就是说,构造函数给对象设置一个值,使随后的各个操作写起来很简单,并能成功地完成工作。这也意味着构造函数常常需要申请资源。如果这件事无法完成,构造函数可以抛出异常,这样我们就可以在创建起对象之前去处理问题。语言和标准库直接支持这一工作方法(E.3.5节)。

要求在抛出异常之前释放资源并将操作对象置于合法状态,也就意味着异常处理的重担将由抛出的函数、在调用链上的函数和处理器共同承担。抛出异常并不是将处理错误变成“其他程序的事情”。释放自己的资源,将操作对象置于某种协调的状态,这些都是抛出和传递异常的函数的职责。除非它们做了这些事情,否则,异常处理器除了试图得体地终止程序之外,就很难再做更多的事了。

E.7 忠告

- [1] 弄清楚你想要什么级别的异常时安全性; E.2节。
- [2] 异常时安全性应该是整体容错策略的一部分; E.2节。
- [3] 为所有的类提供基本保证,也就是说,维持一个不变式,而且不流失资源; E.2节、E.3.2节、E.4节。
- [4] 在可能和可以负担之处提供强保证,使操作或者成功,或者保持所有操作对象不变; E.2节、E.3节。
- [5] 不要从析构函数里抛出异常; E.2节、E.3.2节、E.4节。
- [6] 不要从一个遍历合法序列的迭代器里抛出异常; E.4.1节、E.4.4节。
- [7] 异常时安全性涉及到仔细检查各个操作; E.3节。
- [8] 将模板设计为对异常透明的; E.3.1节。
- [9] 更应该用申请资源的构造函数方式,不要采用`init()`函数; E.3.5节。
- [10] 为类定义一个不变式,使什么是合法状态变得非常清晰; E.2节、E.6节。

- [11] 确保总将对象放在合法状态中，也不要怕抛出异常；E.3.2节、E.6节。
- [12] 保持不变式简单；E.3.5节。
- [13] 在抛出异常之前，让所有操作对象都处于合法状态；E.2节、E.6节。
- [14] 避免资源流失；E.2节、E.3.1节、E.6节。
- [15] 直接表示资源；E.3.2节、E.6节。
- [16] 记住`swap()`有时可以成为复制元素的替代方式；E.3.3节。
- [17] 在可能时依靠操作的顺序，而不是显式地使用try块；E.3.4节。
- [18] 在替代物已经安全生成之前不销毁“老”信息；E.3.3节、E.6节。
- [19] 依靠“资源申请即初始化”技术；E.3节、E.3.2节、E.6节。
- [20] 确保关联容器的比较操作能够复制；E.3.3节。
- [21] 标明关键性数据结构，并为它们定义能够提供强保证的操作；E.6节。

E.8 练习

1. (*1) 列出E.1节的`f()`可能抛出的所有异常。
2. (*1) 回答E.1节例子之后的问题。
3. (*1) 定义一个类**Tester**，让它偶然地从基本操作（例如构造函数）中抛出异常。用**Tester**测试你的标准库。
4. (*1) 找出**vector**构造函数的“糟糕”版本（E.3.1节）中的错误，并写出一个程序使它崩溃。
提示：首先实现**vector**的析构函数。
5. (*2) 实现提供基本保证的简单表。详细论述该表在提供这些保证时对用户的要求。
6. (*3) 实现提供了强保证的简单表。仔细测试这个表，给出为什么人们应该相信它安全的论据。
7. (*2.5) 重新实现11.12节的**String**，使之像标准容器一样安全。
8. (*2) 比较**vector**的赋值和**safe_assign()**的不同版本的运行时间（E.3.3节）。
9. (*1.5) 不用赋值运算符而复制一个分配器（如E.3.3节里改进**operator=()**所需要的那样）。
10. (*2) 给**vector**加入能提供基本保证的单元素和多元素**erase()**和**insert()**操作。
11. (*2) 给**vector**加入能提供强保证的单元素和多元素**erase()**和**insert()**操作。从代价和复杂性方面将这些解与练习10中的解做一个比较。
12. (*2) 写一个**safe_insert()**（E.4.2节），它将一些元素插入一个现存的**vector**（而不是复制到一个临时量）。你必须对元素操作提出哪些要求？
13. (*2) 写一个**safe_insert()**（E.4.2节），它将一些元素插入一个现存的**map**（而不是复制到一个临时量）。你必须对元素操作提出哪些要求？
14. (*2.5) 从大小、复杂性和性能诸方面比较练习12和练习13里的**safe_insert()**与E.4.2节的**safe_insert()**。
15. (*2.5) 写一个只为关联容器使用的、更好的（更简单并更快的）**safe_insert()**。利用特征类写一个**safe_insert()**，使它能自动对各种容器选择最优的**safe_insert()**。提示：19.2.3节。
16. (*2.5) 设法重写**uninitialized_fill()**（19.4.4节、E.3.1节），使之能处理析构函数抛出的异常。能够做到吗？如果能，代价是什么？如果不能，原因是什么？
17. (*2.5) 设法重写**uninitialized_fill()**（19.4.4节、E.3.1节），使之能处理迭代器由于++或--

抛出的异常。能够做到吗？如果能，代价是什么？如果不能，原因是什么？

18. (*3) 从某个不同于标准库的库中取一个容器，检查其文档，弄清它提供了什么样的异常时安全性保证。做一些试验，看看它在抵御由存储分配和用户代码抛出的异常方面的可靠性。将它与相对应的标准库容器做一些比较。
19. (*3) 设法通过忽略出现异常的可能性去优化取自E.3节的**vector**。例如，删除try块等。将它的性能与取自E.3节的版本和某个标准库实现中的**vector**比较。此外，再比较这些不同**vector**的源代码大小和复杂性。
20. (*1) 为**vector**定义不变式（E.3节），带着或者不带**v == 0**的可能性（E.3.5节）。
21. (*2.5) 阅读某个**vector**实现的源代码。看赋值、多元素**insert()**和**resize()**的实现提供了什么保证。
22. (*3) 写一个**hash_map**（17.6节）的版本，使之能像标准库容器一样安全。

索引

知识分为两类，
我们自己知道那个事物，
或者我们知道从何处可以找到有关它的信息。
——Samuel Johnson

- G
 - false 和, 65
 - string 和, 518
 - 常量表达式 (constant-expression), 728
 - 空指针 (null pointer), 730
 - 零, 空 (zero null), 80
- - complex, 598
 - distance(), 486
 - minus, 458
 - negate, 458
 - valarray, 584
 - 迭代器 (iterator), 487
 - 减运算符 (minus operator), 21
- - 迭代器 (iterator), 487
 - 减量运算符 (decrement operator), 112
 - 用户定义运算符 (user-defined operator), 259~260
- ##, 144
- ' 字符文字量 (character literal), 66
- !
 - basic_ios, 542
 - logical_not, 457
 - valarray, 583
- !--
 - bitset, 437
 - complex, 598
 - not_equal_to, 457
 - string, 521
 - valarray, 585
 - 不等运算符 (not equal operator), 21
 - 迭代器 (iterator), 486
 - 生成的 (generated), 415
- # 预定义指令 (preprocessing directive), 711
 - #define, 143
 - #elif, 711
 - #else, 711
 - #endif, 145
 - #error, 712
 - #if, 711
 - #ifdef, 145
 - #ifndef, 192
 - #include, 178
 - #include 保护符 (guard), 192
 - #line, 712
 - #pragma, 712
 - #undef, 145
 - c 字符, 73
 - %
 - modulus, 458
 - valarray, 585
 - 格式符, 574
 - 取模、余数运算符 (modulus operator), 21
 - %%: 二联符, 726
 - %%: 二联符, 726
 - %=, valarray, 584
 - %> 二联符, 726
 - %c 格式 (format), 574
 - %d 格式 (%d format), 574
 - %e 格式 (format), 574
 - %f 格式 (format), 574
 - %G 格式 (%G format), 574
 - %g 格式 (%g format), 574
 - %i 格式 (%i format), 574
 - %n 格式 (format), 574
 - %o 格式 (format), 574
 - %p 格式 (format), 574
 - %s 格式 (format), 574

- 格式 (format), 574
- %X
 - 格式 (format), 574
 - 时间格式 (time format), 793
- %x
 - 格式 (format), 574
 - 日期格式 (date format), 793
- &
 - bitset, 438
 - valarray, 585
 - 按位与运算符 (bitwise and operator), 108
 - 禁止 (prohibiting), 236
 - 引用 (reference), 88
 - 预定义 (predefined), 236
- &&
 - logical_and, 457
 - valarray, 585
 - 逻辑与运算符 (logical and operator), 110
- &=
 - bitset的 (of bitset), 437
 - valarray, 584
- () 和初始式 (and initializer), 75
- *
 - complex, 598
 - multiplies, 458
 - valarray, 585
 - 乘运算符 (multiply operator), 21
 - 迭代器 (iterator), 486
 - 和[], -> 和, 258 ~ 259
- **, 234
- */ 注释结束 (end of comment), 24
- *=
 - complex, 597
 - valarray, 584
- ,
 - 和[], 733
 - 禁止 (prohibiting), 236
 - 预定义 (predefined), 236
 - 运算符 (operator), 110
- .
 - 浮点 (floating-point), 67
 - 成员访问运算符 (member access operator), 91
- . *
 - 到成员的指针 (pointer to member), 371
 - 运算符 (operator), 747
- ..., 省略号, 138
- .c 文件 (file), 179
- .h
 - 头文件 (header), 718
 - 文件 (file), 178
- /
 - complex, 598
 - divides, 458
 - valarray, 585
 - 除运算符 (divide operator), 21
- /*
 - 注释 (comment), 144
 - 注释开始 (start of comment), 24
- //
 - 与C的差异 (difference from C), 713 ~ 714
 - 注释 (comment), 9
- /-
 - complex, 597
 - valarray, 584
- :
 - 标号 (label), 123
 - 派生类 (derived class), 269
 - 算术条件 ?: (arithmetic-if), 119
 - 位域 (bit field), 735
- ::
 - namespace和, 150
 - 和virtual函数, 运算符 (and virtual function, operator), 278
 - 显式限定 (explicit qualification), 741
 - 运算符 (operator), 270
 - 作用域解析运算符 (scope resolution operator), 75
- ::: 到成员的指针 (pointer to member), 371
- ::> 二联符 (digraph), 726
- ;, 分号 (semicolon), 91
- ? :
 - 算术条件 (arithmetic-if), 119
 - 运算符 (operator), 120
- []
 - , 和, 733
 - > 和*和, 258~259
 - bitset, 437
 - map, 426
 - valarray, 582
 - vector的 (of vector), 695
 - 的设计 (of design), 263
 - 迭代器 (iterator), 487
 - 对于string (on string), 515
 - 和insert (), 432
- 反斜线, 726

- 转义字符 (escape character), 726
- \b, 退格, 726
- \f, 换页符 (formfeed), 726
- \n, 换行符 (newline), 726
- \r, 回车符 (carriage return), 726
- \t, 水平制表符 (horizontal tab), 726
- \v, 垂直制表符 (vertical tab), 726
- ^
- bitset, 438
 - valarray, 585
- 按位异或运算符 (bitwise exclusive or operator), 108
- ^=
- bitset的, 437
 - valarray, 584
- _ 字符, 73
- _byname_facet, 775
- _copy后缀 (_copy suffix), 472
- _cplusplus, 183
- _if 后缀 (suffix), 464
- _newhandler, 329
- |
- b_tset, 438
 - valarray, 585
- 按位或运算符 (bitwise or operator), 108
- ||
- logical_or, 457
 - valarray, 585
- 逻辑或运算符 (logical or operator), 110
-
- bitset的, 437
 - valarray, 584
-
- bitset, 437
 - valarray, 584
- 按位补运算符 (bitwise complement operator), 108
- 和析构函数 (and destructor), 217
- +
- complex, 598
 - plus, 458
 - string, 523
 - valarray, 585
- 迭代器 (iterator), 486
- 加运算符 (plus operator), 21
- 用户定义运算符 (user-defined operator), 236
- ++
- 迭代器 (iterator), 486
 - 用户定义运算符 (user-defined operator), 259 ~ 260
- 增量运算符 (increment operator), 112
- +=
- advance(), 487
 - complex, 597
 - string, 522
 - valarray, 584
- 迭代器 (iterator), 487
- 用户定义运算符 (user-defined operator), 236
- 运算符 (operator), 98
- <
- less, 457
 - string, 521
 - template语法 (syntax), 710
 - valarray, 585
 - vector, 406
- 比较 (comparison), 414
- 迭代器 (iterator), 487
- 小于运算符 (less than operator), 21
- <% 二联符, 726
- <: 二联符, 726
- <<
- bitset, 437~438
 - char, 537
 - complex, 538
 - money_put和, 788
 - num_put和, 782
 - ostream, 536
 - streambuf, 564
 - string, 527
 - time_put和, 799
 - valarray, 585
 - virtual, 539
- 插入符 (insert), 535
- 放入 (put to), 535
- 函数指针 (pointer to function), 555
- 实例, Season, 771, 812
- 输出 (output), 41
- 输出运算符 (output operator), 535
- 异常和 (exception and), 782
- 用于输出为什么, (for output why), 535
- 优先级 (precedence), 535
- <<=
- bitset的 (of bitset), 437
 - valarray, 584
- <=
- less_equal, 457
 - string, 521

- valarray, 585
- 迭代器 (iterator), 487
- 生成的 (generated), 415
- 小于等于运算符 (less than or equal operator), 21
- < >, 747
- < >.template, 307
- <algorithm>, 383
- <assert.h>, 384
- <bitset>, 383
- <cassert>, 384
- <cctype>, 384
- <cerrno>, 384
- <cfloat>, 385, 580~581
- <climits>, 385, 580~581
- <ctype>, 385,
- <cmath>, 385, 580~581
- <cssetjmp>, 385
- <csignal>, 385
- <csdarg>, 385
- <csdlo>, 178
- <csdlin>, 383, 385, 482, 509, 530
- <cstring>, 384
- <ctime>, 384
- <ctype.h>, 384
- <wchar>, 384
- <wctype>, 384
- <deque>, 383
- <errno.h>, 384
- <exception>, 338~339, 342~343, 383
- <float.h>, 393
- <fstrean>, 561
- <functional>, 461
- <iomanip>, 384
- <ios>, 536
- <iosfwd>, 534
- <iostream>, 536, 540
 - <istream>和, 540
 - <ostream>和, 536
- <istream>, 384
 - 和<iostream>, 540
- <iterator>, 383
- <limits.h>, 581
- <limits>, 579
- <list>, 383
- <locale.h>, 572
- <locale>, 384
- <rap>, 383
- <math.h>, 385, 580~581
- <memory>, 506
- <new>, 508
- <numeric>, 599
- <ostream>, 384
 - 和<iostream>, 536
- <queue>, 383
- <set>, 383
- <setjmp.h>, 385
- <signal.h>, 385
- <sstream>, 563
- <stack>, 383
- <stdarg.h>, 139
- <stddef.h>, 385
- <stddef>, 385
- <stdexcept>, 384
- <stdib.h>, 383, 385, 482, 509, 530
- <stdio.h>, 384
- <streambuf>, 384
- <string.h>, 528
- <string>, 42, 384
- <stringstream.h>, 577
- <time.h>, 791
- <typeinfo>, 385
- <utility>, 415
- <valarray>, 582
- <vector>, 383
- <wchar.h>, 774
- <wctype.h>, 530
- <wtype.h>, 384
- - map, 428
 - string, 518
 - valarray, 582
 - vector, 396~397
- 继承和 (inheritance and), 273
- 禁止 (prohibiting), 236
- 生成的 (generated), 253
- 用户定义运算符 (user-defined operator), 250
- 预定义 (predefined), 236
- ==
 - complex, 597
 - valarray, 584
- 迭代器 (iterator), 487
- 运算符 (operator), 98
- ==
 - bitset, 437

complex, 598
 equal_to, 457
 string, 521
 valarray, 585
 vector, 406
 等于运算符 (equal), 21
 迭代器 (iterator), 487
 相等, 没有 (equality without), 415
 用户定义运算符 (user-defined operator), 471 ~ 472
 >
 greater, 457
 string, 521
 valarray, 585
 大于运算符 (greater than operator), 21
 迭代器 (iterator), 487
 和>>, 710
 生成的 (generated), 415
 ->
 成员访问运算符 (member access operator), 91
 迭代器 (iterator), 487
 和*和[], 258~259
 用户定义运算符 (user-defined operator), 257 ~ 259
 -> *
 成员访问运算符 (member access operator), 91
 到成员的指针 (pointer to member), 371
 浮点 (floating-point), 67
 运算符 (operator), 747
 >=
 greater_equal, 457
 string, 521
 valarray, 585
 大于等于运算符 (greater than or equal operator), 21
 迭代器 (iterator), 487
 生成的 (generated), 415
 >>
 >和, 710
 bitset, 437~438
 char, 540
 complex, 547
 istream, 540
 money_get和, 789
 num_get和, 783
 string, 527
 time_get, 794, 795
 valarray, 585
 函数指针 (pointer to function), 555
 取出 (get from), 535

实例, Season, 771, 812

输入cin, 44, 100~101

提取符 (extractor), 535

>>=

bitset的, 437

valarray, 584

i, true和, 65

16位字符 (16bit character), 512

-1, 727

-1和size_t, 398

8位char (8bit char), 512

A

Aarhus, 473
 abort(), 193
 abs(), 580~581, 587
 abs(), valarray, 586
 accumulate(), 599
 acos(), valarray, 586
 Ada, 9
 adjacent_find(), 464
 adjacent_difference(), 600
 adjustfield, 551
 advance() 和+=, 487
 Algol68, 9
 allocate(), 500
 allocator, 默认的 (default), 500
 allocator_type, 393, 425
 always_noconv(), codecvt, 809
 and
 关键字 (keyword), 726
 运算符 &&, 逻辑 (operator &, logical), 110
 运算符 &, 按位 (operator &, bitwise), 108
 and_eq关键字 (keyword), 726
 Annemarie, 83
 ANSI
 C, 12
 C++, 10
 any(), 438
 append(), string, 522
 apply(), valarray, 584
 app接在文件之后 (append to file), 562
 arg(), 598
 argc, main() argv, 105
 argv argc, main(), 105
 ASCII, 511
 字符集 (character set), 66, 530

asctime(), 791
 asin(), 581
 valarray, 586
 asm汇编 (asm assembler), 706
 Assert(), 658
 assert(), 658
 assign()
 char_traits, 512
 string, 519
 vector, 396~397
 Assoc实例, 255
 AT&T贝尔实验室, 10
 at(), 47
 out_of_range和, 342
 对string, 516
 对vector, 396
 atan(), 581
 valarray, 586
 atan2(), 581
 valarray, 586
 atc, 562
 atexit()
 和析构函数 (and destructor), 193
 和异常 (and exception), 340
 atof(), 530
 atoi(), 520
 atol(), 530
 auto, 738
 auto_ptr, 326

B

back(), 396
 back(), queue的, 421
 back_inserter(), 50, 490
 back_inserter_iterator, 490
 bad(), 542
 bad_alloc, 115
 和new, 340
 缺少 (missing), 720
 异常 (exception), 508
 bad_cast, 365
 bad_exception, 336
 bad_type和typeid(), 342
 badbit, 543
 base(), 499
 basefield, 551~553
 Basic, 636

basic_filebuf, 类, 571
 basic_ifstream, 561
 basic_ios, 534, 542
 !, 543
 格式状态 (format state), 534
 流状态 (stream state), 534
 basic_iostream, 560
 格式化 (formatting), 533
 basic_istream, 540
 basic_istringstream, 563
 basic_ofstream, 561
 basic_ostream, 535~536
 basic_ostringstream, 563
 basic_streambuf, 567
 缓冲 (buffering), 533
 basic_string, 513
 basic_stringstream, 514
 begin(), 515
 const_iterator, 514
 const_pointer, 514
 const_reference, 514
 const_reverse_iterator, 514
 difference_type, 514
 end(), 515
 iterator, 514
 pointer, 514
 rbegin(), 515
 reference, 514
 rend(), 515
 reverse_iterator, 514
 size_type, 514
 trait_type, 514
 value_type, 514
 成员类型 (member type), 513
 BCPL, 9
 before(), 369
 beg, seekdir和, 565
 begin()
 basic_string, 294, 426
 迭代器 (iterator), 515
 Bi, 452
 bidirectional_iterator_tag, 489
 binary 二进制模式 (binary mode), 562
 binary_function, 456
 binary_negate, not2()和, 462
 binary_negate, 459
 binary_search(), 477

bind1st(), 458, 460
 bind2nd(), 458 ~ 459
 binder1st, 458, 460
 binder2nd, 458 ~ 459
 BinOp, 452
 BinPred, 452
 bitand关键字 (keyword), 108
 bitor关键字, 726
 bitset, 436
 !, 437
 &, 438
 &=, 437
 [], 439
 /, 437
 ^-, 436
 |, 437
 |=, 437
 ~, 437
 <<, 437~438
 <=, 437
 =, 437
 >>, 437~438
 >=, 437
 flip(), 437
 reset(), 437
 set(), 437
 操作 (operation), 437
 构造函数 (constructor), 436
 和enum, 436
 和set, 436
 和vector<bool>, 436
 和位域 (and bit field), 436
 输出 (output), 426
 输入 (input), 428
 bitset().invalid_argument和, 342
 BLAS, 587
 Blixen, Karen, 2
 Bomb实例, 818
 Bool, 65
 的vector, 405
 的输出 (output of), 536
 的输入 (input of), 540
 转换到 (conversion to), 730
 boolalpha, 538
 boolalpha(), 557
 break, 99, 118 ~ 119
 case和, 120
 语句 (statement), 104

bsearch(), 482
 和异常, 840
 buffer, 296
 实例, 647

C

C

//, 与C的差异 (difference from), 713 ~ 714
 ANSI, 12
 enum, 差异 (difference from), 715
 locale, 764
 sizeof, 差异 (difference from), 713
 struct名字, 差异 (struct name, difference from),
 714~717
 struct作用域, 差异 (struct scope, difference from),
 714~717
 void*赋值, 差异 (void* assignment, difference from),
 714~717
 标准库 (standard library), 815
 参数类型, 差异 (argument type, difference from),
 713~714
 差异 (difference from), 713
 程序员 (programmer), 13
 初始化和goto, 差异 (initialization and goto, difference
 from), 714~717.
 函数调用, 差异 (function call, difference from),
 713~714
 函数定义, 差异 (function definition, difference
 from), 713~714
 函数和异常 (function and exception), 339
 和C++, 7, 9, 13, 177
 和C++ locate, 768
 和C++、学习 (learning), 6
 和C++混合 (mixing), 631
 和C++兼容性 (compatibility), 12, 713~714
 和C++全局locate (global locate), 806
 和异常 (and exception), 341
 宏, 差异 (macro, difference from), 714~717.
 声明和定义, 差异 (declaration and definition, difference
 from), 714~717.
 输入和输出 (input and output), 573
 数组初始式, 差异 (array initializer, difference from),
 714~717.
 跳过初始化, 差异 (jump past initialization, difference
 from), 715
 隐式的int, 差异 (int implicit, difference from), 715
 与C连接 (linkage to), 182

- 与类 (and Classes), 9
- 作用域, 差异 (scope, difference from), 713~714
- c_array, 439
- c_str(), 519
- C++, 19
 - ANSI, 10
 - C和 (C and), 7, 9, 13, 177
 - ISO, 10
 - locale, C和 (locale, C and), 768
 - 标准化 (standardization), 10
 - 程序员 (programmer), 13
 - 大型程序和 (large program and), 8
 - 的设计 (design of), 7, 9
 - 的使用 (use of), 12
 - 的误用 (misuse of), 636
 - 的性质 (properties of), 635
 - 的逐步采纳 (gradual adoption of), 630
 - 读音 (pronunciation), 9
 - 风格的下标 (style subscript), 592
 - 功能分解和 (functional decomposition and), 637
 - 过程式程序设计和 (procedural programming and), 636
 - 兼容性, C和 (compatibility, C and), 12, 713~714
 - 教学和 (teaching and), 12
 - 库, 第一个 (library, first), 603
 - 特征总结 (feature summary), 716
 - 学习 (learning), 6, 630
 - 学习C和 (learning C and), 6
 - 意义 (meaning), 9
 - 引进 (introducing), 630
 - 与C混合 (mixing C and), 631
 - 逐步推进地学习方式 (gradual approach to learning), 6
- C++的性质 (properties of C++), 635
- cache实例, 207
- call_from_C(), 342
- callC(), 342
- calloc(), 509
- capacity(), vector, 405
- Car实例, 676
- CASE, 636
- case和break, 120
- catch, 167
 - 每一个异常 (every exception), 48
 - 通过引用 (by reference), 320
 - 通过值 (by value), 320
 - 一切 (all), 323
- catch (...), 48
- catch一切 (all, catch), 323
- ceil(), 581
- cerr, 537
 - 初始化 (initialization), 556
 - 和clog (and clog), 549
- char, 66, 69
 - <<, 536
 - >>, 540
 - 8位 (8-bit), 512
 - get(), 544
 - signed, 728
 - unsigned, 728
 - 的输出 (output of), 536
 - 的输入 (input of), 540
 - 里的位数 (bits in), 578~579
 - 字符类型 (character type), 65
- char*, 专门化和 (char*, specialization and), 307
- CHAR_BIT, 581
- char_traits, 512
 - assign(), 513
 - char_type, 512
 - compare(), 513
 - copy(), 513
 - eof(), 513
 - eq(), 513
 - eq_int_type(), 512
 - find(), 512
 - get_state(), 513
 - int_type(), 513
 - length(), 512
 - lt(), 513
 - move(), 513
 - not_eof(), 513
 - off_type, 513
 - pos_type, 513
 - state_type, 513
 - to_char_type(), 513
 - to_int_type(), 513
- char_traits<char>, 512
- char_traits<wchar>, 513
- char_type, 536
- char_typechar_traits, 512
- Checked_iter实例 (Checked_iter example), 495
- Checked实例 (Checked example), 499
- Cin, 540
 - >>, 输入 (>>, input), 44, 100~101
- cout和 (cout and), 549
 - 初始化 (initialization), 556

- 的值 (value of), 245~246
- class, 199, 201
 - facet, 768
 - locale, 762
 - namespace和 (namespace and), 741
 - string, 260
 - struct和 (struct and), 208
 - template和 (template and), 310~311
 - typename和 (typename and), 749
 - union和 (union and), 737
 - 不是 (not a), 619
 - 层次结构 (hierarchy), 34, 424~425
 - 层次结构漫游 (hierarchy navigation), 362
 - 成员 (member), 265
 - 成员访问 (access to member), 744
 - 抽象 (abstract), 622
 - 抽象结点 (abstract node), 677
 - 定义 (definition), 201
 - 格 (lattice), 346
 - 公共基类 (universal base), 391
 - 和概念 (and concept), 199
 - 基类与派生 (base and derived), 35, 646
 - 具体 (concrete), 210, 215, 671
 - 具体结点 (concrete node), 677
 - 门类 (kind of), 670
 - 嵌套 (nested), 261
 - 声明 (declaration), 201
 - 随机数 (random number), 602
 - 像函数的 (function-like), 455
 - 协助 (helper), 327
 - 叶 (leaf), 677
 - 用户定义类型 (user-defined type), 199
 - 自立的 (free-standing), 643
- classic() locale, 571
- clear(), 542
 - failure和 (failure and), 342
 - map, 431
 - vector
 - 和异常 (and exception), 834
- clock(), 789
- clock_t, 789
- CLOCKS_PER_SEC, 789
- Clock实例 (Clock example), 354
- clog, 536
 - cerr和 (cerr and), 549
 - 初始化 (initialization), 556
- clone(), 378
- close(), 562
 - 消息 (messages), 810
- Clu, 10
- Club_eq, 457
- Cmp, 302
- Cobol, 636
- codecvt
 - always_noconv(), 809
 - encoding(), 809
 - facet, 807
 - in(), 808
 - length(), 809
 - max_length, 809
 - out(), 809
 - unshift(), 809
- codecvt_base result, 807
- collate
 - compare(), 776~838
 - do_compare(), 776
 - do_hash(), 776
 - do_transform(), 776
 - facet, 776
 - hash(), 776
 - transform(), 776
- collate_byname, 779
- combine(), 766
- compare()
 - char_traits, 513
 - collate, 776~838
 - string, 520
- complex, 57, 237
 - , 597
 - !-, 597
 - *, 597
 - *, 597
 - /, 597
 - /=, 597
 - +, 597
 - +=, 597
 - <<, 538
 - ==, 597
 - ==, 597
 - >>, 547
 - cos(), 598
 - cosh(), 598
 - exp(), 598
 - log(), 598

- log10(), 598
 - pow(), 598
 - sin(), 598
 - sinh(), 598
 - sqrt(), 598
 - tanh(), 598
 - 和complex<> (and complex<>), 597
 - 和异常 (and exception), 840
 - 输出 (output), 598
 - 输入 (input), 598
 - 数学函数 (mathematical functions), 598
 - 运算 (operations), 598
 - 转换 (conversion), 598
 - complex<>, complex 和, 598
 - compl关键字 (compl keyword), 726
 - conj(), 597
 - const, 85
 - C风格字符串和 (C-style string and), 81
 - 成员 (member), 222
 - 成员函数 (member function), 204
 - 迭代器 (iterator), 393, 449
 - 函数, 探查 (function, inspector), 620
 - 和连接 (and linkage), 176
 - 和重载 (and overloading), 529
 - 强制去掉 (casting away), 367
 - 物理的和逻辑的 (physical and logical), 206
 - 指向 (pointer to), 85~86
 - 指针 (pointer), 85
 - const_cast, 117, 207
 - const_iterator, 514
 - basic_string, 514
 - const_mem_fun_ref_t, 458, 461
 - const_mem_fun_t, 458, 461
 - const_mem_fun1_ref_t, 459, 461
 - const_mem_fun1_t, 458, 461
 - const_pointer, 393
 - basic_string, 514
 - const_reference, 393, 425
 - basic_string, 514
 - const_reverse_iterator, 393, 425
 - basic_string, 514
 - construct(), 501
 - continue, 104
 - 语句 (statement), 104
 - copy(), 468, 513
 - char_traits, 513
 - copy_backward(), 468
 - copy_if() 不是标准 (copy_if() not standard), 469
 - copyfmt(), 552
 - copyfmt_event, 573
 - copyfmt_event, copyfmt(), 573
 - cos(), 581
 - complex, 598
 - cosh(), 581
 - complex, 598
 - valarray, 586
 - count(), 429
 - 在map里 (in map), 429
 - count_if(), 54, 465
 - cout, 536
 - 初始化 (initialization), 556
 - 和cin (and cin), 550
 - 输出 (output), 41
 - cowboy实例 (cowboy example), 681
 - CRC卡片 (CRC card), 618
 - cshift(), 584
 - ctype
 - facet, 804
 - is(), 804
 - narrow(), 805
 - scan_is(), 804
 - scan_not(), 804
 - tolower(), 805
 - toupper(), 804
 - widen(), 805
 - ctype_base, 803
 - cur, seekdir和
 - curr_symbol(), moneypunct, 786
 - currying化 (currying), 460
 - cvt_to_upper 实例, 809
 - C风格 (C-style)
 - 初始化, 构造函数和 (initialization, constructor and), 241
 - 错误处理 (error handling), 581
 - 分配 (allocation), 509
 - 函数和算法 (function and algorithm), 462
 - 强制 (cast), 117
 - 强制, 贬斥 (cast, deprecated), 716
 - 字符串, string和 (string, string and), 511
 - 字符串和const (string and const), 81
- ## D
- data(), 519
 - Date_format实例, 797
 - Date_in实例, 799

`date_order()`, 802
`dateorder`, 795
`Date` 实例, 210, 792, 796
`DBL_MINEXP`, 580
`deallocate()`, 501
`dec`, 558
`decimal_point()`, 779
 `moneyprint`, 786
`declaration`, 703
`declarator`, 706
`default`, 98
`delete`
 `delete[]` 和, 224
 大小和 (size and), 374
 和废料收集 (and garbage collection), 738
 数组 (array), 224
 运算符 (operator), 115
`delete()`, operator, 508
`delete[]`, 115
 和 `delete`, 224
`delete[]()`, operator, 508
`delete_ptr()` 实例, 470
`denorm_min()`, 580
`deque`
 和异常 (and exception), 832
 双端队列 (double-ended queue), 420
`destroy()`, 502
`difference_type`, 514
 `basic_string`, 514
`digits`, 579
`digits10`, 579
`distance()` 和 `-`, 489
`div()`, 581
`div_t`, 581
`divides`, 458
`do_compare()`, `collate`, 776
`do_hash()`, `collate`, 776
`do_it()` 例子, 680
`do_transform()`, `collate`, 776
`double`, 67
 `long`, 67
 输出 (output), 552
`do` 语句 (do statement), 122
`draw_all()` 实例, 460
`Duff` 设施 (Duff's device), 126
`dynamic_cast`, 361 ~ 362
 `bad_cast` 和, 342
 的实现 (implementation of), 363

 的使用 (use of), 677
 对引用 (to reference), 363
 和 `static_cast`, 366
 和多态性 (and polymorphism), 363
 和歧义性 (and ambiguity), 366
`dynamic_cast` 和, 365

E

`eatwhite()`, 546
`eback()`, 568
`EDOM`, 581
`egptr()`, 568
`eliminate_duplicates()` 实例, 472
`else`, 120
`Employee` 实例, 269
`empty()`, 411
 `string`, 527
`encoding()`, `codecvt`, 809
`end()`, 426
 `basic_string`, 515
 迭代器 (iterator), 393
`end, seekdir` 和
`endl`, 558
 和 `std`, 556
`ends`, 558
`enum`, 69
 `bitset` 和, 436
 `sizeof`, 71
 成员 (member), 222
 和整数 (and integer), 70
 用户定义运算符和 (user-defined operator and), 236
 与C的差异 (difference from C), 713~714
 转换, 无定义 (conversion, undefined), 70
`EOF`, 574
`eof()`, 542
 `char_traits`, 513
`eofbit`, 543
`ep_ptr()`, 568
`epsilon()`, 580
`eq()`, `char_traits`, 513
`eq_int_type()`, `char_traits`, 513
`equal()`, 466
`equal_range()`, 478
 在 `map` 里, 429
`equal_to ==`, 457
`Erand`, 602
`ERANGE`, 581

erase()
 map, 431
 vector, 401
 和异常 (and exception), 832
 在string里 (in string), 525
 errno, 581
 event, 573
 event_callback, 573
 exceptions(), 782
 exit(), 194
 exp(), 581
 complex, 598
 exp(), valarray, 586
 explicit构造函数 (explicit constructor), 254
 export, 182
 expression, 699
 Expr实例, 377
 extern, 176
 Extract_officers实例, 463

F

fabs(), 581
 facet
 _byname, 775
 class, 768
 codecvt, 807
 collate, 776
 ctype, 804
 locale和, 762
 messages, 810
 money_get, 788
 money_put, 788
 moneypunct, 785
 nur_get, 783
 nur_put, 780
 numpunct, 779
 put() 迭代器 (iterator), 781
 Season_io, 用户定义 (user-defined), 771
 time_get, 795
 time_put, 792
 标识符id (identifier id), 770
 标准 (standard), 774
 访问 (access to), 770
 加入locale (to locale, adding), 766
 类别 (category), 774~775
 生存时间 (lifetime of), 769
 使用 (use of), 770
 用户定义 (user-defined), 774
 fail, 542
 failbit, 543
 failed, 782
 failure和clear(), 342~343
 falsename(), 780
 false和0, 65
 filebuf, 571
 fill(), 554
 fill_n(), 474
 Filter实例, 688
 find(), 464
 char_traits, 513
 在map里, 429
 在string里, 524
 find_end(), 467
 find_first_not_of() 在string, 524
 find_first_of(), 464
 在string, 524
 find_if(), 464
 find_last(), 394
 find_last_of() 在string, 524
 fixed, 553
 fixed(), 558
 flags(), 552
 flags() bitset, 437
 float, 67
 里的位 (bits in), 578~579
 输出 (output), 552
 float_denorm_style, 580
 float_round_style, 580
 floatfield, 553
 floor(), 581
 FLT_RADIX, 580
 flush, 558
 flush(), 565
 fmod(), 581
 For, 452
 for
 语句 (statement), 121
 语句, 声明 (statement, declaration in), 122
 for(;;), 98
 for_each(), 463
 Form实例, 559
 Fortran
 风格的下标 (style subscript), 592
 向量 (vector), 587

forward_iterator_tag, 489
for 语句初始化表达式 (*for*-statement initializer), 721
 frac_digits()
 money_punct, 785
 货币量的 (of monetary amount), 787
 free(), 509
 frexp(), 581
 friend, 249, 745, 747
 friend 的 (of friend), 745
 template 和, 746
 函数 (function), 249
 和成员 (and member), 250
 类 (class), 249
 派生和 (derived and), 746
 声明 (declaration), 249
 front(), 418
 queue 的, 421
 front_insert_iterator, 490
 front_inserter(), 490
 fstream, 561

G

gbump(), 568
 gcount(), 544
 generate(), 474
 generate_n(), 474
 get(), 566
 char, 546
 messages, 810
 money_get, 788
 num_get, 783
 函数 (function), 665
 get_allocator(), 406
 出自串, 527
 get_date(), 795
 get_monthname(), 800
 get_state(), char_traits, 513
 get_temporary_buffer(), 507
 get_time(), 795
 time_get, 795
 get_weekday(), 800
 get_year(), 795
 getchar(), 574
 getline(), 544
 放入串, 527
 getloc(), 572
 global(), locale, 768
 gmtime(), 791
 good(), 542
 goodbit, 543
 goto
 非局部 (nonlocal), 318
 和初始式 (and initializer), 122
 和异常 (and exception), 122
 与C在初始化和*goto*上的差异 (difference from C initialization and), 715
 gp_ptr(), 568
 greater >, 457
 greater_equal >=, 457
 grouping(), 779
 moneypunct, 786
 gslice, 595
 gslice_array, 595

H

handle_ioexception(), 782
handler, 异常 (handler, exception), 711
 has_denorm, 580
 has_denorm_loss, 580
 has_facet(), 770
 has_infinity, 580
 has_quiet_NaN, 580
 has_signaling_NaN, 580
 hash(), collate, 778
 hash_map, 440
 resize(), 444
 表示 (representation), 441
 查找 (lookup), 442
 散列函数 (hash function), 440
 删除 (delete from), 443
 相等 (equality), 440
 Hello, world! 实例, 40
 hex, 558
 Histogram, 404

I

I/O, 53, 533, 547, 567, 570, 600
 sentry, 550
 的实现 (implementation of), 534
 迭代器和 (iterator and), 53
 对象 (object), 677
 非缓冲的 (unbuffered), 570
 缓冲 (buffering), 568

- 可扩充 (extensible), 533
- 宽字符 (wide character), 536
- 类型安全 (type safe), 535
- 流和异常 (stream and exception), 839
- 设计 (design), 533
- 系统的组织 (system, organization of), 534
- 异常 (exception), 547
- id, facet标识符 (identifier), 770
- identity() 实例, 470
- IEC-559, is_iec, 559, 580
- if
 - switch 和, 120
 - 语句 (statement), 119
- ifstream, 561
- ignore(), 544
- imag(), 597 ~ 598
- imbue(), 567, 570, 572, 685
 - imbue_event, 573
 - iostream locale, 761
- imbue_event, imbue(), 573
- implicit_cast, 299
- In, 452
- in 为读打开 (open for reading), 562
- in(), codecvt, 808
- in_avail(), 569
- includes(), 479
- indirect_array, 597
- infinity(), 580
- init()
 - 函数 (function), 827
 - 和构造函数 (and constructor), 830
- inline
 - 成员函数 (member function), 210
 - 函数 (function), 129
 - 和连接 (and linkage), 177
- inner_product(), 599
- inplace_merge(), 478
- input_iterator_tag, 489
- insert(), 49
 - [] 和, 432
 - list, 835
 - map, 835
 - string, 522
 - vector, 835
 - 多元素 (multiple-element), 836
 - 和异常 (and exception), 832
- insert_iterator, 490
- inserter(), 490
- int, 66, 69
 - 里的位数 (bits in), 578~579
 - 输出的位数 (output bits of), 438
 - 隐含, 与C的差异 (implicit, difference from C), 713~714
 - 最大的 (largest), 578
 - 最小的 (smallest), 578
- TNT_MAX, 580
- int_type, 536
- int_type(), char_traits, 513
- internal, 555
- internal(), 558
- invalid_argument和bitset(), 342
- io_obj实例, 678
- io_state, 719
- iocopy() 实例, 543
- ios, 719
- ios_base, 551
 - 格式化状态 (format state), 534
- iosbase::init实例, 562
- iostate, 719
- iostream, 560
 - locale, imbue(), 761
 - locale和, 774
 - locale在...里, 762
 - sentry, 550
 - 和异常 (and exception), 839
- is(), ctype, 804
- is_exact, 580
- is_iec559, IEC-559, 580
- is_integer, 578
- is_modulo, 580
- is_signed, 579
- is_specialized, 579
- isalnum(), 530
 - 和locale, 806
- isalpha(), 101, 530
 - 和locale, 806
- isdigit(), 580
 - 和locale, 530
- Iseq, 454
- iseq()实例, 530, 805
- isgraph(), 530
 - 和locale, 806
- islower(), 530
 - 和locale, 806

ISO C++, 10
 ISO-I4882, 10
 ISO-4217, 786
 ISO-646, 725
 isprint(), 530
 和 locale, 806
 ispunct() 和 locale, 806
 isspace(), 530
 空白 (whitespace), 102
 isspace 和 locale, 806
 istream, 540
 >>, 540
 的赋值 (assignment of), 536
 复制 (copy), 536
 和迭代器 (and iterator), 492
 和异常 (and exception), 839
 istream_iterator, 53, 492
 istreambuf_iterator, 492
 istringstream, 563
 istrstream, 577
 isupper(), 530
 和 locale, 806
 isxdigit(), 530
 和 locale, 806
 iter_swap(), 476
 iterator, 514
 basic_string, 514
 iterator_category, 487
 iterator_traits, 487
 itoa(), 139
 ltor, 387
 Ival_box 实例, 282
 Ival_slider, 355
 iword(), 572

J~K

JIS, 807
 Kernighan 和 Ritchie, 575
 key_comp(), 429
 key_compare, 425
 key_type, 425
 Knuth Donald, 626

L

L: 宽字符文字量 (wide-character literal), 66
 labs(), 581

Latin-1, 512
 ldexp(), 581
 ldiv(), 581
 ldiv_t, 581
 left, 555
 left(), 558
 length()
 char_traits, 513
 codecvt, 809
 string, 516
 string 的, 527
 less, 456
 <, 457
 less_equal <=, 457
 less_than, 459
 lexicographical_compare(), 序列的 (of sequence), 481
 Link, 350
 Liskov 替换 (substitution), 651
 Lisp, 636
 List, 387
 list, 48
 insert(), 835
 merge(), 478
 merge() 算法和 (algorithm and), 417
 merge() 稳定 (stable), 834
 push_back(), 400
 remove(), 419
 remove_if(), 419
 sort() 稳定 (stable), 417
 unique(), 419
 和异常 (and exception), 832
 双向链 (doubly-linked), 416
 元素访问 (element access), 418
 locale, 572
 C, 764
 class, 762
 classic(), 768
 C 和 C++, 768
 C 和 C++, 全局 (global), 806
 global(), 768
 imbue() iostream, 761
 isalnum() 和, 806
 isalpha() 和, 806
 iscntrl() 和, 806
 isdigit() 和, 806
 isgraph() 和, 806
 islower() 和, 806

isprint() 和, 806
 ispunct() 和, 806
 isspace() 和, 806
 isupper() 和, 806
 isxdigit() 和, 806
 POSIX, 572
 String, 777
 初始 (initial), 768
 的生存时间 (lifetime of), 780
 改变 (changing), 768
 概念 (concept), 759
 格式化信息 (format information), 534
 构造函数 (constructor), 766
 和 facet, 759
 和 iostream, 774
 加入 facet (adding facet to), 766
 经典 (classic) C, 764
 类别 (category), 762
 名字 (name), 762~764
 名字字符串 (name string), 765
 默认 (default), 768
 偏爱的 (preferred), 764
 设计准则 (design criteria), 582
 设置 (setting), 768
 是不可变的 (is immutable), 768
 文化习俗 (cultural preference), 759
 修改 (modify), 768
 用于字符串比较 (used for string comparison), 768
 在 iostream 里, 762
 locale(), 572
 localtime(), 791
 Lock_ptr 实例, 326
 log(), 581
 complex, 598
 valarray, 586
 log10(), 581
 complex, 598
 valarray, 586
 logical_and &&, 457
 logical_not, 457
 !, 457
 logical_or ||, 457
 long, 66
 double, 67
 lower_bound(), 478
 在 map 里, 429
 lt(), char_traits, 513

M

Machiavelli, Niccolò, 198
 main(), 193
 argv argc, 105
 和初始化 (and initialization), 193
 异常和 (exception and), 48
 make_heap(), 480
 make_pair(), 427
 malloc(), 509
 map, 425
 [], 427
 -, 428
 clear(), 526
 count(), 429
 equal_range(), 429
 erase(), 431
 find(), 429
 insert(), 835
 lower_bound(), 431
 upper_bound(), 431
 成员类型 (member type), 425
 的使用 (use of), 677
 迭代器 (iterator), 426
 赋值 (assignment), 428
 构造函数 (constructor), 428
 和异常 (and exception), 832
 里的比较 (comparison in), 428
 下标 (subscripting), 427
 修改 (modify), 431
 元素访问 (element access), 427
 mapped_type, 425
 Marian, 72
 mask, 字符 (character), 803
 masks_array, 596
 math_container 实例, 309
 Matrix, 747
 实例, 252
 max(), 481
 valarray, 584
 max_element(), 序列的 (of sequence), 481
 max_exponent, 580
 max_exponent10, 580
 max_length(), codecvt, 809
 max_size(), 396
 string 的, 527
 nbstate_t, 774

mem_fun(), 458, 461
 mem_fun_ref(), 458, 461
 mem_fun_ref_t, 458, 461
 mem_fun_t, 458, 461
 mem_fun1_ref_t, 458, 461
 mem_fun1_t, 458, 461
 member-declaration, 708
 memchr(), 510~511
 memcmp(), 510
 memcpy(), 509
 memmove(), 509
 memset(), 510
 merge(), 478
 算法和 (algorithm and) list, 478
 稳定 (stable), list, 417
 messages
 close(), 810
 facet, 811
 get(), 810
 open(), 810
 messages_base, 810
 min(), 579
 valarray, 584
 min_exponent, 580
 min_exponent, 10, 580
 minus -, 458
 mismatch(), 466
 mktime(), 791
 ML, 9
 modf(), 581
 modulus %, 458
 monetaryfrac_digits(), 787
 money_base, 785
 money_get
 and >>, 789
 facet, 788
 get(), 788
 money_punct frac_digits(), 786
 money_put
 facet, 788
 put(), 788
 和 <<, 788
 moneypunct
 curr_symbol(), 786
 decimal_point(), 786
 facet, 785
 grouping(), 787

neg_format(), 786
 negative_sign(), 786
 pattern, 786
 pos_format(), 786
 positive_sign(), 786
 thousands_sep(), 786

Money实例, 784

move(), char_traits, 513

multimap, 433

multiplies *, 458

multiset, 435

mutable, 207

My_messages实例, 810

My_money_io实例, 787

My_punct实例, 779

N

namespace, 741

rel_ops, 415

std, 41

using, 164

别名 (alias), 159

别名, 重新打开 (alias re-open), 166

成员声明和定义 (member declaration and definition),
150~151

从中选择 (selection from), 162

的作用 (purpose of), 162

和 ::, 151

和类, 742

和重载 (and overloading), 164

连接与 (linkage and), 184

名字, 长 (name, long), 160

名字, 短 (name, short), 159

名字查找 (name lookup), 159

全局 (global), 741

缺少 (missing), 719

是开放的 (is open), 165

无名的 (unnamed), 159

限定的名字 (qualified name), 151

协助函数和 (helper function and), 214

运算符和 (operators and), 236

重新打开 (re-open), 166

组合 (composition), 162

narrow(), 805

ctype, 805

NDEBUG, 658

neg_format(), moneypunct, 786

negate-, 458
 negative_sign(), moneypunct, 786
 new
 bad_alloc和, 342
 大小和 (size and), 374
 放置 (placement), 228
 和数组 (and array), 375
 和异常 (and exception), 509
 异常和 (exception and), 328
 运算符 (operator), 115
 new()
 operator, 375
 放置 (placement), 509
 new[], 508
 new[](), operator, 375
 new_handler, 508
 next_permutation(), 482
 Nicholas, 43
 noboolalpha() 557
 Nocase, 414
 none(), 437
 norm(), 598
 noshowbase(), 557
 noshowpoint(), 557
 noshowpos(), 557
 noskipws(), 557
 not_eof(), char_traits, 513
 not_equal_to. =, 457
 not_eq关键字 (keyword), 726
 not2(), 459
 和binary_negate, 462
 nothrow, 508
 分配器 (allocator), 720
 not1(), 459
 和unary_negate, 462
 not关键字 (keyword), 726
 nouppercase(), 557
 npos, 517
 nth_element(), 477
 NULL, 385
 num_get
 facet, 783
 get(), 783
 和>>, 783
 num_put
 facet, 781
 put(), 781

和<<, 781

numeric_limits, 579
 numpunct facet, 779
 n元运算符 (n-ary operators), 593

O

Object, 391
 实例, 371
 oct, 551~553
 oct(), 558
 off_type, 566
 char_traits, 513
 ofstream, 561
 Op, 452
 open(), messages, 810
 openmode, 562
 operator
 delete(), 115, 508
 delete[](), 375
 new(), '508
 函数表 (functions, list of), 234
 operator, 709
 operator(), 256
 operator[], 255
 operator=, vector, 824
 or
 关键字 (keyword), 726
 运算符I, 按位 (operatorI, bitwise), 111
 运算符II, 按位 (operatorII, logical), 110
 or_eq关键字 (keyword), 778
 oseq() 实例, 491
 ostream, 565
 <<, 536
 put(), 537
 template和, 536
 write(), 537
 的赋值 (assignment of), 536
 复制 (copy), 536
 和streambuf, 565
 和迭代器 (and iterator), 492
 和缓冲区 (and buffer), 565
 和异常 (and exception), 839
 ostream_iterator, 492
 ostreambuf_iterator, 492
 ostreambuf迭代器 (iterator), 492
 ostrstream, 563
 ostrstream, 577

out, 452
 out(), codecvt, 809
 out. 为读打开 (open for writing), 562
 out_of_range, 396
 out_of_range, string, 517
 out_of_range和at(), 342
 output_iterator_tag, 489
 overflow(), 571
 overflow_error和to_ulong(), 342

P

partial_sort(), 477
 partial_sort_copy(), 477
 partial_sum(), 600
 partition(), 479
 pattern, moneypunct, 786
 pbackfail(), 570
 pbase(), 568
 pbump(), 568
 peek(), 566
 pnone_book实例, 46
 Plane实例, 640
 plus i, 458
 pointer, 487
 basic_string, 514
 pointer_to_binary_function, 461
 pointer_to_unary_function, 461
 polar(), 598
 Pool_alloc分配器 (allocator), 505
 Pool实例, 503
 pop()
 priority_queue, 423
 queue, 422
 stack, 421
 pop_back(), 400
 和异常 (and exception), 832
 pop_front(), 419
 和异常 (and exception), 832
 pop_heap(), 480
 pos_format(), moneypunct, 786
 pos_type, 566
 char_traits, 513
 positive_sign(), moneypunct, 786
 POSIX
 locale, 572
 格式修饰符 (format modifier), 793
 pow(), 581

 complex, 598
 valarray, 587
 pptr(), 568
 precision(), 553
 Pred, 452
 prev_permutation(), 482
 printf(), 573
 priority_queue
 pop(), 423
 push(), 423
 top(), 423
 堆和 (heap and), 480
 和堆 (and heap), 425
 实现 (implementation), 423
 private, 357
 基类 (base class), 651
 基类 (base), 651
 基类, 覆盖 (base, override), 647~648
 private: , 209
 protected, 357
 private, public, 742~744
 成员 (member), 259~360
 构造函数 (constructor), 769
 基 (base), 360
 基类 (base class), 651
 界面 (interface), public和, 568
 ptr, 312
 ptr_fun(), 458, 461
 ptrdiff_t, 109, 385
 pubimbue(), 569
 public, 357
 protected private, 742~744
 和保护界面 (interface), 568
 public protected, 742~744
 public: , 201
 pubseedkoff()
 pubseekpos()
 pubsetbuf(), 569
 pubsync(), 569
 push()
 priority_queue, 423
 queue, 422
 stack, 421
 push_back(), 400
 list, 834
 vector, 834
 和realloc(), 400

和异常 (and exception), 832
 push_back(), vector, 825
 push_front(), 49, 418~419
 和异常 (and exception), 832
 push_heap(), 480
 put()
 money_put, 788
 num_put, 781
 ostream, 536
 time_put, 792
 迭代器 (iterator), facet, 781
 put(), 792
 putback(), 566
 pword(), 572

Q

qsort(), 142
 和异常 (and exception), 840
 queue
 deque, 双端 (double-ended), 420
 优先 (priority), 423
 queue
 back(), 422
 front(), 422
 pop(), 422
 push(), 422
 消息 (message), 423
 quiet_NaN(), 580

R

Ran, 452
 rand(), 随机数 (random number), 602
 RAND_MAX, 602
 Randint, 602
 random_access_iterator_tag, 489
 random_shuffle(), 475
 Range实例, 684
 Rational实例, 665~656
 raw_storage_iterator, 508
 rbegin(), 426
 basic_string, 515
 迭代器 (iterator), 394
 rdbuf(), 567
 rdstate(), 542
 rdstr(), 563
 read(), 544

readsome(), 566
 real(), 597~598
 realloc(), 509
 push_back() 和, 400
 rebind, 502
 的使用 (use of), 502
 reference, 487
 basic_string, 514
 到二进制位 (to bit), 436
 register, 706
 register_callback(), 573
 reinterpret_cast, 228
 rel_ops.namespace, 416
 remove(), 474
 list, 419
 和异常 (and exception), 832
 remove_copy_if(), 474
 remove_if(), 474
 list, 419
 和异常 (and exception), 832
 rend(), 426
 basic_string, 515
 迭代器 (iterator), 394
 replace(), 473
 在string里, 525
 replace_copy(), 473
 replace_copy_if(), 473
 replace_if(), 473
 reserve(), vector, 404~405
 reset() bitset, 437
 resetiosflags(), 558
 resize(), 46
 hash_map, 444
 string的, 527
 valarray, 583
 vector, 404
 和迭代器 (and iterator), 446
 result, codecvt_base, 807
 return
 void表达式 (of void expression), 132
 函数值 (function value), 132
 其他方式 (alternative), 318
 通过引用 (by reference), 132~133
 通过值 (by value), 132~133
 return, 132
 return_temporary_buffer(), 508

reverse(), 475
reverse_copy(), 475
reverse_iterator, 491
rfind()在string里, 524
right, 555
right(), 558
Ritchie, Kernighan, 575
rotate(), 475
rotate_copy(), 475
round_error(), 580
RTTI, 362
 的实现 (implementation of), 363
 的使用 (use of), 371
 的误用 (misuse of), 392
runtime_error异常 (exception), 764

S

Saab实例, 639
safe_assign(), 825
Safe实例, 817
Satellite, 346
shumpc(), 569
scan_is(), ctype, 804
scan_not(), ctype, 804
scientific, 553
scientific(), 558
scrollbar实例, 652
search(), 467
search_n(), 467
season
 <<实例, 771, 812
 >>实例, 771, 812
 实例, 771
season_10, 用户定义 (user-defined) facet, 771
seekdir
 和beg, 565
 和cur, 565
 和end, 565
 寻找的方向 (direction of seek)
seekg()
 的方向 (direction of), 565
seekoff()
seekp()
 的方向 (direction of), 565
 设置位置 (set position), 565
seekpos()
sentry
 I/O, 550
 ostream, 550
set, 435
 bitset和, 436
 shape* 的, 311
set()
 bitset, 437
 函数 (function), 665
set_controller实例, 687~688
set_difference(), 480
set_intersection(), 479
set_new_handler(), 508
set_symmetric_difference(), 480
set_terminate(), 338
set_unexpected(), 338
set_union(), 479
setbase(), 558
setbuf(), 569
setf(), 555
setfill(), 558
setg(), 568
setiosflags(), 558
setp(), 568
setprecision(), 557~558
setstate(), 543
setw(), 558
set实例, 674
sgetc(), 569
sgetn(), 569
Shape
 实例 (example), 677
 实例 (example), 28
 实例 (坏) (example (bad)), 371
shape*, 的集合 (set of), 311
shift(), 584
short, 66
showbase, 552
showbase(), 557
showmanyc(), 570
showpoint, 551
showpoint(), 557
showpos, 551
showpos(), 557
signaling_NaN(), 580
signed
 char, 727

- unsigned整数转换 (integer conversion), 730
- 类型 (type), 66
- Simula, 9, 34
- Simula 风格的容器 (Simula-style container), 391
- sin(), 581
 - complex, 598
 - valarray, 587
- sinh(), 581
 - complex, 598
 - valarray, 587
- size(), 437
 - string, 517
 - string的, 527
 - valarray, 584
- size_t, 385
 - l和, 398
- size_type, 425
 - basic_string, 514
- sizeof, 68
 - enum, 71
 - 的结果 (result of), 109
 - 与C的差异 (difference from C), 713~714
- skipws, 550
- skipws(), 557
- slice, 587
- slice_array, 589
- slice_icer实例, 589
- Smalltalk, 636
 - 风格 (style), 371
- Smalltalk风格的容器 (Smalltalk-style container), 391
- sm manip, 557
- snextc(), 569
- sort(), 476
 - 和异常 (and exception), 832
 - 实例 (example), 298
 - 稳定 (stable), list, 417
- sort_heap(), 480
- splice(), 417
 - 和异常 (and exception), 832
- sputbackc(), 569
- sputc(), 569
- sputn(), 569
- sqrt(), 581
 - complex, 598
 - valarray, 587
- srand(), 602
- stable_partition(), 479
- stable_sort(), 477
- stack
 - pop(), 421
 - push(), 421
 - top(), 421
 - 上溢 (overflow), 422
 - 实现 (implementation), 421~422
 - 下溢 (underflow), 422
- stack实例, 24
- state_type, char_traits, 513
- statement, 702
- static
 - template的成员 (member of template), 747
 - 贬斥 (deprecated), 716
 - 不合适宜 (anachronism), 177
 - 成员 (member), 374
 - 成员函数 (member function), 249
 - 存储, 局部 (store, local), 224
 - 对象 (object), 218
 - 局部 (local), 129
- static_cast, 116, 142~143
 - dynamic_cast和, 366
- std
 - namespace, 41
 - 操控符和 (manipulator and), 556
- std::, 41
- STL, 58
 - 迭代器 (iterator), 391
 - 容器 (container), 391
- Storable实例, 352
- str(), 562
- strcat(), 528
- strchr(), 528
- strcmp(), 528
- strcpy(), 528
- strncpy(), 528
- streambuf, 384, 492, 564
 - <<, 565
 - ostream和, 565
 - 迭代器 (iterator), 492
 - 和字符缓冲区 (and character buffer), 565
- streamoff, 536
- streamsize, 536
- strftime(), 793
- stride(), 587
- string, 563~564
- string, 42, 513

- !=, 521
- [] 在string上, 515
- +, 523
- +=, 522
- <, 521
- <<, 528
- <=, 521
- ~, 518
- =, 521
- >, 521
- >=, 521
- >>, 528
- append(), 522
- assign(), 519
- class, 263
- compare(), 521
- empty(), 527
- erase(), 525
- find(), 524
- find_first_not_of(), 524
- find_last_of(), 524
- get_allocator(), 524
- getline(), 528
- insert(), 522
- length(), 517
- locale, 777
- out_of_range, 517
- replace(), 525
- rfind(), 524
- size(), 517
- swap(), 528
- unsigned, 514
- 比较 (comparison), 521
- 错误 (error), 517
- 的at(), 516
- 的length(), 527
- 的max_size(), 527
- 的resize(), 527
- 的size(), 527
- 的substr(), 526
- 的范围检查 (range check of), 515
- 的下标 (subscripting of), 516
- 的隐式转换 (implicit conversion of), 520
- 迭代器 (iterator), 515
- 赋值 (assignment), 518
- 构造函数 (constructor), 516
- 和0, 518
- 和C风格字符串 (and C-style string), 519
- 和数组 (and array), 519
- 和异常 (and exception), 839
- 空 (empty), 516
- 流 (stream), 563~564
- 拼接 (concatenation), 522~523
- 设计 (design), 511
- 输出 (output), 527
- 输入 (input), 527
- 算法和 (algorithm and), 515
- 文字量 (literal), 262
- 序列 (sequence), 511
- 用户定义类型的 (of user-defined type), 514
- 转换 (conversion), 520
- 作为容器 (as container), 435
- string_numput实例, 781
- stringbuf, 571
- stringstream, 563
- String实例, 293
- strlen(), 528
- strncat(), 528
- strncpy(), 528
- strcpy(), 528
- strpbrk(), 529
- strchr(), 528
- strstr(), 528
- struct
 - 和class, 209
 - 聚集 (aggregate), 91
 - 麻烦, 708
 - 名字, 与C的差异 (name, difference from C), 715
 - 作用域, 与C的差异 (scope, difference from C), 715
- substr(), string的, 526
- substring实例, 526
- sum, 455
- sum(), valarray, 584
- sungetc(), 569
- swap(), 824
 - string, 528
 - 和异常 (and exception), 836
- swap_ranges(), 476
- switch, 98
 - 和if, 120
- 基于枚举 (on enumeration), 70
- 语句 (statement), 119
- sync(), 570
- sync_with_stdio(), 573

T

- Table实例, 217
- `tan()`, `valarray`, 586
- `tanh()`, 581
- `complex`, 598
- `valarray`, 586
- `Task`, 350
- `tellig()`, 566
- `tellp()` 得到位置 (get position), 565
- `template`, 747
 - `<>`, 307
 - 包含 (inclusion), 312
 - 参数 (argument), 296
 - 参数 (parameter), 296
 - 参数, `template`作为 (parameter, `template as`), 747
 - 参数, 非类型 (parameter, non-type), 296
 - 参数, 函数 (argument, function), 299
 - 参数, 默认 (argument, default), 721
 - 参数, 推断 (argument, deducing), 748
 - 参数, 显式 (argument, explicit), 299
 - 参数, 依赖于 (argument, dependency on), 752
 - 参数化 (parameterization), 621
 - 成员 (member), 294
 - 的`static`成员 (member of), 747
 - 定义的上下文 (definition, context of), 751
 - 分别编译 (separate compilation), 312
 - 复制赋值和 (copy assignment and), 311
 - 复制构造函数和 (copy constructor and), 312
 - 函数 (function), 298
 - 函数, `virtual` (function, `virtual`), 311
 - 和`class`, 311
 - 和`friend`, 747
 - 和`ostream`, 536
 - 和宏 (and macro), 754
 - 和继承 (and inheritance), 309
 - 和通用型程序设计 (and generic programming), 292
 - 继承和 (inheritance and), 312
 - 类层次结构和 (class hierarchy and), 308
 - 例子, 成员 (example, member), 312
 - 名字约束 (name binding), 751
 - 缺少成员 (missing member), 720
 - 实例化 (instantiation), 751
 - 实例化, 显式 (instantiation, explicit), 757
 - 实例化的上下文 (instantiation, context of), 751
 - 实例化指令 (instantiation directive), 757
 - 语法 `<` (syntax `<`), 710
 - 源代码 (source code), 312
 - 在设计里 (in design), 663~664
 - 重载, 函数 (overloading, function), 300
 - 专门化 (specialization), 304
 - 作为`template`参数 (as `template parameter`), 747
 - 作为限定词 (as `qualifier`), 750
- `template`, 的使用 (use of), 679
- `template-declaration`, 710
- `terminate()`, 338
- `terminate_handler`, 338
- `test()`, 437
- `this`, 248
 - 自引用 (self-reference), 205
- `thousands_sep()`
 - `moneypunct`, 786
 - 风格字符 (separator character), 780
- `throw`, 7
- `tie()`, 548
- `time()`, 791
- `time_base`, 795
- `time_get`
 - `facet`, 795
 - `get_time()`, 795
 - 和`>>`, 794, 796
- `time_put`
 - `facet`, 792
 - 和`<<`, 798
- `time_t`, 时间表示 (time representation), 789
- `tinyness_before`, 580
- `Tiny` 实例, 245
- `tm`, 时间表示 (time representation), 789
- `to_char_type()`, `char_traits`, 513
- `to_int_type()`, `char_traits`, 513
- `to_ulong()`, 438
 - `overflow_error`和, 342
- `tolower()`, `cctype`, 805
- `top()`
 - `priority_queue`的, 423
 - `stack`的, 421
- `toupper()`, 521
 - `cctype`, 804
- `traits_type`, 536
 - `basic_string`, 514
- `transform()`, 469
 - `collate`, 778
- `traps`, 580
- `truenam()`, 780

true和1, 65
 truncate 截断文件 (truncate file), 562
 try, 167
 try-块 (block), 168
 作为函数体 (as function body), 332
 type_info, 368
 typedef, 76
 typeid(), 368
 bad_typeid和, 342
 typename, 393
 和class, 750

U

uflow(), 570
 unary_function, 456
 unary_negate, 459
 not1()和, 462
 uncaught_exception(), 333
 underflow(), 570
 unexpected(), 334
 unexpected_handler, 338
 unset(), 566
 Unicode, 807
 uninitialized_copy(), 507
 和异常 (and exception), 839
 uninitialized_fill(), 821
 和异常 (and exception), 839
 uninitialized_fill_n(), 507
 和异常 (and exception), 839
 union, 735
 成员 (member), 738
 成员对象 (member object), 218
 构造函数和 (constructor and), 229
 和class, 737
 匿名的 (anonymous), 736
 无名的 (unnamed), 736
 析构函数和 (destructor and), 229
 指针和 (pointers and), 738
 unique(), 471
 list, 419
 和异常 (and exception), 832
 unique_copy(), 471
 unistbuf, 551
 UNIX, 7, 12
 Unsafe实例, 817
 unsetf(), 551
 unshift(), codecvt, 809

unsigned
 char, 727
 string, 514
 类型 (type), 66
 整数转换 (integer conversion), signed, 730
 update()实例, 834
 upper_bound(), 478
 在map里, 429
 uppercase, 551
 uppercase(), 557
 Urand, 602
 use_facet(), 770
 using
 namespace, 163
 namespace, using与, 741
 using namespace, 741

V

va_arg(), 139
 valarray, 586
 valarray, 582
 -, 584
 !, 584
 !=, 586
 %, 586
 %=: 584
 &, 586
 &&, 586
 &=: 584
 *, 586
 *=, 584
 /, 586
 /=, 584
 [], 583
 ^, 586
 ^=: 584
 |, 586
 |=: 586
 |=, 584
 ~, 584
 +, 586
 +=, 584
 <, 586
 <<, 586
 <<=: 584
 <=: 586
 =, 583

- =, 584
- =, 586
- >, 586
- >=, 586
- >>, 586
- >>=, 584
- abs(), 586
- acos(), 586
- apply(), 584
- asin(), 586
- atan(), 586
- atan2(), 586
- cos(), 586
- cosh(), 586
- exp(), 586
- log(), 586
- log10(), 586
- max(), 584
- min(), 584
- pow(), 586
- resize(), 584
- sin(), 586
- sinh(), 586
- size(), 584
- sqrt(), 586
- sum(), 584
- tan(), 586
- tanh(), 586
- 操作 (operations), 584
- 的长度 (length of), 597
- 迭代器 (iterator), 589
- 范围检查 (range check), 584
- 赋值 (assignment), 583
- 构造 (construction), 582
- 和vector和数组 (and array), 582
- 和数组 (and array), 583
- 和异常 (and exception), 840
- 输出 (output), 587
- 输入 (input), 587
- 数学函数 (mathematical functions), 586
- 下标 (subscripting), 583
- 作为容器 (as container), 436
- valarray的长度 (length of valarray), 597
- value_comp(), 429
- value_compare, 429
- value_type, 514
- value_type, basic_string, 514
- Vec, 范围检查 (range checking), 360
- Vector, 747
 - <, 406
 - =, 397
 - ==, 406
 - assign(), 397
 - bool的, 407
 - capacity(), 405
 - clear()
 - erase(), 401
 - insert(), 835
 - push_back(), 834
 - push_back(), 825
 - reserve(), 405
 - resize(), 405
 - vector的, 732
 - 成员类型 (member type), 392
 - 的[], 396
 - 的vector, 732
 - 对vector的at(), 396
 - 构造函数 (constructor), 398
 - 和数组, valarray和 (and array, valarray and), 582
 - 和异常 (and exception), 832
 - 减小容量 (decrease capacity of), 405
 - 实例, 683
 - 输入到 (input into), 400
 - 运算符= (operator=), 825
 - 增加大小 (increase size of), 405
 - 增加容量 (increase capacity of), 405
- vector, 396~397, 400, 695
- vector_base, 822
- vector<bool>, 407
 - bitset和, 436
- Vehicle实例, 644
- virtual, 31
 - <<, 539
 - template函数 (function), 311
 - 的返回类型 (return type of), 377
 - 构造函数 (constructor), 288
 - 函数 (function), 276
 - 函数, 纯的 (function, pure), 278
 - 函数, 定义的 (function, definition of), 276
 - 函数, 运算符::和 (function, operator :: and), 278
 - 函数参数类型 (function argument types), 276
 - 函数的例子 (function, example of), 568~569

函数的实现 (function, implementation of), 32

基类 (base class), 352

基类, 覆盖 (base class, override from), 356

基类, 构造函数和 (base, constructor and), 353

没有虚函数的派生 (derive without), 681~684

输出函数 (output function), 539

析构函数 (destructor), 284

void, 69

表达式, 的返回 (expression, return of), 132~133

指向void的指针 (pointer to), 90

void*

赋值, 与C的差异 (assignment, difference from C), 715

专门化和 (specialization and), 292

void*(), 543

void*(), 运算符 (operator)

volatile, 707

W

wcerr, 536

wchar_t, 65~66

wcin, 540

wcout和, 549

wclog, 536

wcout, 536

和wcin, 549

wfilebuf, 571

wistream, 562

while语句, 122

widen(), 567

ctype, 805

wioth(), 554

输入的 (of input), 543

wifstream, 562

Window实例, 354

wiostream, 561

wistream, 540

wistream, 563

wofstream, 562

wostream, 536

wostream, 563

write(), ostream, 537

ws, 558

wstreambuf, 571

wstring, 513

wstringbuf, 571

wstringstream, 563

X ~ Y

X3J16, 10

xalloc(), 572

xor_eq关键字 (keyword), 726

xor关键字 (keyword), 726

xsetn(), 570

xsetn(), 570

Year, 254

(按汉语拼音排序)

A

安全性 (safety)

方便性与 (convenience vs.), 741

和析构函数, 异常时 (and destructor, exception), 817

条件, 异常时 (condition, exception), 817

异常时 (exception), 816

按成员复制 (memberwise copy), 253

按对象的数据 (per-object data), 506

按类的数据 (per-type data), 506

按位 (bitwise)

补运算符 ~, 111

或运算符 |, 111

逻辑运算符, 111

异或运算符 ^, 111

与运算符 &, 111

B

八进制 (octal), 66

输出 (output), 555

白痴 (moron), 630

半开序列 (half-open sequence), 453

包含, template (inclusion, template), 312

包含目录, 标准 (include directory, standard), 178

包容 (containment), 648

和继承 (and inheritance), 649

包装器 (wrapper), 684

保持一致的声明 (keeping consistent declarations), 178

保护 (protection), 202

单位 (unit of), 661

保留的名字 (reserved names), 73

保证 (guarantee)

nothrow, 817

标准 (standard), 724

和权衡 (and tradeoff), 834

基本 (basic), 817

- 强 (strong), 817
- 容器 (container), 830
- 异常 (exception), 817
- 总结, 异常 (summary, exception), 833
- 贝尔实验室, (Bell Laboratories, AT&T), 10
- 本国的 (national)
 - 常规 (conventions), 572
 - 字符集 (character), 725
- 比较 (comparison)
 - <, 414
 - string, 520
 - 串 (string), 776
 - 大小写不敏感 (case-insensitive), 521
 - 默认的 (default), 414
 - 相等和 (equality and), 405
 - 需求 (requirement), 414
 - 用户定义 (user-supplied), 414
 - 用于串比较的 locale (locale used for string), 768
 - 运算符, 运算符 (operator, operator), 21
 - 在 map 里 (in map), 428
- 闭包 (closure), 595
- 编码员和设计师 (coders and designers), 610
- 编入程序里的关系 (programmed-in relationship), 654
- 编译 (compilation)
 - template 分别 (template separate), 312
 - 单位 (unit of), 175
 - 分别 (separate), 24, 175~176
- 编译单位 (translation unit), 175
- 编译时多态性 (compile-time polymorphism), 309~310
- 编译时间, 头文件和 (compile time, header and), 186
- 贬斥 (deprecated)
 - C 风格强制 (C-style cast), 716
 - static, 716
 - 非 const 字符串常量 (non-const string literal), 716
 - 特征 (feature), 714~717
- 变大小的对象 (variably-sized object), 217
- 变量 (variable)
 - 参数的数目 (number of argument), 138
 - 局部变量的构造函数 (constructor for local), 219
 - 临时的 (temporary), 227
 - 全局 (global), 177
 - 全局变量的构造函数 (constructor for global), 224~225
- 遍历 (traversal), 54
- 标点 (punctuation)
 - 货币值 (of monetary amount), 785
 - 数 (number), 436
- 标号 (label)
 - : , 118
 - 作用域 (scope of), 122
- 标识符 (identifier), 73
 - id. facet, 770
 - 的意义 (meaning of), 749
- 标志位, 操纵 (flag manipulation), 552
- 标准 (standard)
 - facet, 774
 - 包含目录 (include directory), 178
 - 保证 (guarantee), 724
 - 操纵符 (manipulator), 557
 - 货币符号 (currency symbol), 786
 - 库 (libraries), 615
 - 库 (library), 163
 - 库, C (library, C), 528
 - 库, 加入 (library, adding to), 386
 - 库, 缺少 (library, missing), 719
 - 库的组织 (library organization), 383
 - 库功能 (library facilities), 58, 379~380
 - 库和异常 (library and exception), 838
 - 库容器 (library container), 392
 - 库设计 (library design), 380~381
 - 库算法 (library algorithms), 56
 - 库头文件 (library header), 178
 - 库谓词 (library predicate), 457
 - 库准则 (library criteria), 381
 - 数学函数 (mathematical functions), 580
 - 提升 (promotion), 729
 - 异常 (exception), 342
 - 组件 (component), 627
- 标准化, C++ (standardization, C++), 10
- 标准前的实现 (pre-standard implementation), 717~718
- 表 (list)
 - 操作 (operation), 401
 - 容器的 (of containers), 383
 - 运算符函数的 (of operator functions), 234
 - 表达式 (expression)
 - 常量 (constant), 728
 - 条件 (conditional), 120
 - 完整 (full), 227
- 表示 (representation)
 - hash_map, 441
 - 容器的 (of container), 413
 - 转换字符 (converting character), 807
- 表示存储 (representing memory), 822
- 别名 (alias)
 - namespace, 158

重新打开, namespace (re-open, namespace), 166
 补运算符~, 按位 (complement operator ~, bitwise), 111
 捕捉一切 (catch all), 323
 不变式 (invariant), 656
 构造函数和 (constructor and), 827
 和简单性 (and simplicity), 827
 和异常 (and exception), 815
 检查 (checking), 657
 简单 (simple), 827
 异常和 (exception and), 827
 不等于运算符 (equal operator) !=, 21
 不检查的访问 (unchecked access), 396
 不可变的, locale (immutable, locale is), 768
 不抛出保证 (nothrow guarantee), 817
 不是类 (not a class), 619
 不同关注点 (separation of concerns), 609
 布局, 数组 (layout, array), 588
 步骤, 设计 (steps, design), 615
 部分的 (partial)
 构造 (construction), 818
 排序 (sort), 476
 专门化 (specialization), 306

C

采纳C++, 逐步 (adoption C++, gradual), 630
 参考文献, 设计 (bibliography, design), 631
 参数化 (parameterization)
 策略 (policy), 664
 和依赖性 (and dependency), 621
 模板 (template), 621
 参数化多态性 (parametric polymorphism), 310
 操控符 (manipulator)
 标准 (standard), 557
 带参数 (with argument), 556
 和s-d, 556
 和作用域 (and scope), 556
 输出 (output), 555
 输入 (input), 556
 用户定义 (user-defined), 558

操作 (operation)

bitset, 437
 表 (list), 401
 的效率 (efficiency of), 412
 迭代器 (iterator), 486
 对容器 (on container), 412
 前端 (front), 419
 自然 (natural), 672

操作 (operations)

class的操作集合 (set of class), 211
 complex, 597
 valarray, 587
 对结构 (on structure), 92
 对引用 (on references), 88
 向量 (vector), 587
 选择 (selecting), 620

测试 (testing), 625

为测试而设计 (design for), 625

策略参数化 (policy parameterization), 664

层次结构 (hierarchy), 642

class, 34, 424~425
 传统的 (traditional), 280
 对象 (object), 656
 界面 (interface), 622
 类 (class), 280, 309
 流 (stream), 561
 漫游, class (navigation, class), 362
 设计, 类 (design, class), 280
 异常 (exception), 342
 重组类 (reorganization of class), 621

插接类比 (plug analogy), 639

插入, 复写和 (insertion, overwriting vs), 490

插入符, << (inserter, <<), 536

查尔斯·达尔文 (Darwin, Charles), 606

查找 (lookup)

hash_map, 442
 namespace名 (name), 236

差异 (difference)

与C, 713~714
 与C++, 713~714
 与C enum, 715
 与C int, 隐含 (implicit), 715
 与C sizeof, 713~714
 与C struct 名字 (name), 714~717.
 与C void* 赋值 (assignment), 714~717.
 与C参数类型 (argument type), 713~714
 与C初始化和goto (initialization and goto), 714~717.
 与C函数调用 (function call), 713~714
 与C函数定义 (function definition), 713~714
 与C宏 (macro), 714~717
 与C声明和定义 (declaration and definition), 714~717.
 与C数组初始式 (array initializer), 714~717.
 与C跳过初始化 (jump past initialization), 715
 与C作用域 (scope), 713~714

长namespace名字 (long namespace name), 160

长期 (longer term), 614

长期 (term, longer), 614

常量 (constant)

表达式 (expression), 728

成员 (member), 222

类中的枚举符 (enumerator as in-class), 222

时间 (time), 412

在类中定义 (in-class definition of), 222

常量表达式 (constant-expression), 728

超类 (superclass), 270

和子类 (and subclass), 35

超末端一个 (one-beyond-last), 453

成功的大型系统 (successful large system), 623

成员 (member)

*, 指针 (pointer to), 372

::*, 指针 (pointer to), 372

->*, 指针 (pointer to), 372

class, 263

class, 访问 (access to), 744

const, 223

enum, 222

friend和, 250

protected, 259~360

static, 374

template, 293

template, 缺少 (missing), 720

template的, static, 747

template实例, 312

union, 737

常量 (constant), 222

初始化 (initialization), 221

初始化, 异常和 (initialization, exception and), 332

初始化, 引用 (initialization, reference), 224

初始化的顺序 (initialization, order of), 221

初始式 (initializer), 221

对象 (object), 218

对象 (object), union, 218

访问 (access to), 742

访问运算符 (access operator) ->, 263

访问运算符 (access operator) ->, 263

公用类 (public class), 201

函数 (function), 212

函数 (function), const, 205

函数 (function), inline, 210

函数 (function), static, 248

函数, 算法和 (function, algorithm and), 460

函数适配器 (function adapter), 460

或者基类 (or base), 649

或者指针 (or pointer), 648

基类的私用 (of base class, private), 272

类, 前向声明 (class, forward declaration of), 263

类的构造函数 (constructor for class), 220

类型 (type), basic_string, 513

类型 (type), map, 425

类型 (type), vector, 392

派生类的 (of derived class), 270

私用类 (private class), 201

异常和 (exception and), 818

引用 (reference), 649

与非成员运算符 (and nonmember operators), 237

指向函数成员的指针 (function, pointer to), 372

指向数据成员的指针 (pointer to data), 746

乘运算符 (multiply operator) *, 21

程序 (program), 40, 699

大型 (large), 186~189

的大小 (size of), 7

的划分 (partitioning of), 187

的结构 (structure of), 7

的逻辑结构 (logical structure of), 176

的物理结构 (physical structure of), 176

非C++ (non-C++), 194

和C++, 大 (and C++, large), 8

计时 (timing a), 789

开始 (start), 191

终止 (termination), 193

程序的划分 (partitioning of program), 187

程序设计 (programming), 14

template和通用型 (and generic), 292

的目标 (purpose of), 609

范型 (paradigm), 20

风格 (style), 20

风格的技术语言 (styles technique language), 6

过程性 (procedural), 20

和C++, 过程性 (procedural), 636

面向对象 (object-oriented), 268

模块化 (modular), 23

设计和 (design and), 608

通用型 (generic), 292, 663~664

语言 (language), 14

语言, 设计语言和 (language, design language and), 641

作为人的活动 (as a human activity), 609

程序设计语言, 通用 (programming-language, general-purpose), 19

- 程序员 (programmer)
 - C, 13
 - C++, 13
- 程序员, 消除 (programmers, elimination of), 641
- 重点, 实例和 (emphasis, examples and), 4
- 重叠的序列 (overlapping sequences), 468
- 重复的关键码 (duplicate key), 433
- 重复的基类 (replicated base class), 350
- 重命名虚函数 (renaming virtual function), 682
- 重新打开 (re-open)
 - namespace, 166
 - namespace别名 (alias), 166
- 重新抛出 (re-throw), 338
- 重用 (reuse), 627
 - 具体类型的 (of concrete type), 673
 - 设计 (design), 623
- 重载 (overload)
 - 返回类型和 (return type and), 135
 - 解析 (resolution), 134
 - 解析, 手工 (resolution, manual), 135~136
 - 作用域和 (scope and), 135
- 重载 (overloading)
 - const和, 529
 - namespace和, 164
 - 函数 (function) template, 300
 - 运算符重载的例子 (example of operator), 346
- 重载的 (overloaded)
 - 函数名 (function name), 133
 - 运算符 (operator), 215
- 抽取符, >> (extractor, >>), 536
- 抽象 (abstraction)
 - 的层次 (levels of), 643
 - 后 (late), 388
 - 类和 (classes and), 643
 - 数据 (data), 26
- 抽象的 (abstract)
 - 类 (class), 621
 - 迭代器 (iterator), 386
 - 和具体的 (and concrete), 675
 - 结点class (node class), 677
 - 类 (class), 278
 - 类, 类层次结构和 (class, class hierarchy and), 289
 - 类和设计 (class and design), 283~284
 - 类型 (type), 27, 30, 63, 109, 392, 502
- 抽象的层次 (levels of abstraction), 643
- 初始locale (initial locale), 768
- 初始化 (initialization), 218
 - cerr, 560
 - cin, 560
 - clog, 560
 - cout, 560
 - main() 和, 192
- 成员 (member), 221
 - 成员的初始化顺序 (order of member), 221
 - 赋值和 (assignment and), 253
 - 构造函数和C风格 (constructor and C-style), 241
 - 和goto, 与C的差异 (and goto, difference from C), 715
 - 和清理 (and cleanup), 323
 - 和异常 (and exception), 836
 - 基类的 (of base class), 272
 - 结构的 (of structure), 92
 - 库 (library), 563
 - 全局的 (of global), 192
 - 引用成员 (reference member), 224
 - 引用的 (of reference), 88
 - 用字符串初始化数组 (of array by string), 80~81
 - 与C风格跳过的差异 (difference from C jump past), 715
 - 运行时 (run-time), 193
 - 资源申请即 (resource acquisition is), 325
- 初始式 (initializer)
 - () 和, 75~76
 - for-语句, 721
 - goto和, 123
 - 成员 (member), 221
 - 列表, 构造函数和 (list, constructor and), 241
 - 默认 (default), 76
 - 数组 (array), 80~81
 - 在类内 (in-class), 222
- 除法运算符 / (divide operator), 21
- 处理 (handling)
 - 错误 (error), 500
 - 异常 (exception), 842
- 传递多维数组 (passing multidimensional array), 733
- 传统的层次结构 (traditional hierarchy), 280
- 创建 (creation)
 - 对象 (object), 216
 - 对象的局部化 (localization of object), 287
- 创建的依赖性 (create dependency), 653~654
- 创新 (innovation), 630
- 垂直制表符 (vertical tab) \v, 726
- 纯的 (pure)
 - virtual函数 (function), 278
 - 面向对象 (object-oriented), 642
- 词法规则 (lexical conventions), 695

从namespace选择 (selection from namespace), 162

从空白开始 (starting from scratch), 622

存储 (memory)

表示 (representing), 822

动态 (dynamic), 30, 392, 639

堆 (heap), 738

堆栈 (stack), 738

管理 (management), 738

管理, 容器 (management, container), 501

管理, 用户定义的实例 (management, example of user-defined), 346

管理, 自动 (management, automatic), 738

耗尽 (exhaustion), 841

缓冲区 (buffer), 507

静态 (static), 738

流失 (leak), 841

碎片 (fragmentation), 740

未初始化 (uninitialized), 506

自动 (automatic), 738

存储 (storage)

类 (class), 218

原始 (raw), 502

存储 (store)

动态 (dynamic), 30

堆 (heap), 115

局部 (local) static, 224

自由 (free), 114, 218, 220

错误 (error)

string, 516

报告 (reporting), 167

处理 (handling), 499

处理, C风格 (handling, C-style), 580~581

处理, 多层次 (handling, multilevel), 341

分析 (analysis), 624

恢复 (recovery), 842

连接 (linkage), 177

其他处理方式 (handling alternative), 172

设计 (design), 624

序列 (sequence), 453

序列和 (sequence and), 453

循环和 (loop and), 463

异常和 (exception and), 48, 122, 328

运行时 (run-time), 312

值域 (range), 581

状态 (state), 816

作用域 (domain), 581

D

打开 (open)

名字空间 (namespace is), 165

为读, 输入 (for reading, in), 562

为写, 输出 (for writing, out), 562

打开文件 (opening of file), 561

大 (large)

程序 (program), 186~189

程序和C++ (program and C++), 8

字符集 (character set), 727

大-O记法 (big-O notation), 412

大O记法 (notation), 412

大小 (size)

vector, 增加 (increase), 405

程序的 (of program), 7

和delete, 374

和new, 374

结构的 (of structure), 92

数的 (of number), 579

尾数的 (of mantissa), 579

序列的, 改变 (of sequence, change), 468

指针的 (of pointer), 68

字符串的 (of string), 132

大小写不敏感的比较 (case insensitive comparison), 521

大于 (greater)

大于等于运算符 >=, 21

大于运算符 >, 21

代理 (proxy), 687

代码 (code)

膨胀, 抑止 (bloat, curbing), 306

统一性 (uniformity of), 620

异常时安全的 (exception-safe), 822

代码的统一性 (uniformity of code), 671

单位 (unit)

保护的 (of protection), 226

编译的 (of compilation), 175

分配的 (of allocation), 80

设计的 (of design), 661

寻址的 (of addressing), 80

单引号 (single quote) \', 726

单元的类比 (units analogy), 639

倒换 (shuffle), 475

等价, 类型 (equivalence, type), 94

等于运算符 -= (equal operator -=), 21

低级语言 (low-level language), 7

递归 (recursion), 132

递归的 (recursive)

函数, 异常和 (function, exception and), 333

下降分析 (decent parser), 97

第一个 (first)

C++库 (library), 603

元素 (element), 396

点 (point)

定义的 (of definition), 753

声明的 (of declaration), 75

实例化的 (of instantiation), 754

点积 (dot product), 600

调用 (call)

函数 (function), 128

析构函数的显式 (of destructor, explicit), 228

引用 (by reference), 88, 225, 248, 263, 649, 800

运算符 (operator), 254~255

值 (by value), 130

迭代器 (iterator), 51, 386

~, 486

~, 486

!=, 486

*, 486

[], 486

+, 486

++, 486

+=, 486

<, 486

<=, 486

-=, 486

==, 486

>, 486

->, 486

>=, 486

begin(), 395

const, 393, 449

end(), 395

facet put(), 781

istream和, 492

map, 426

ostreambuf, 492

ostream和, 492

rbegin(), 394

rend(), 394

resize()和, 443

STL, 391

streambuf, 492

string, 515

valarray, 589

操作 (operation), 486

抽象 (abstract), 391

反向 (reverse), 491

非法 (invalid), 486

合法 (valid), 486

和I/O, 53

和序列 (and sequence), 486

和异常 (and exception), 837

检查的 (checked), 495

类别 (category), 489

流 (stream), 492

命名习惯 (naming convention), 452

前向 (forward), 486

前向和输出 (forward and output), 486

容器 (container), 412

容器和 (container and), 394

实现 (implementation), 52

输出 (output), 486

输入 (input), 486

双向 (bidirectional), 486

随机访问 (random-access), 486

通过迭代器读 (read through), 486

通过迭代器写 (write through), 486

用户定义 (user-defined), 495

定位 (position)

二进制位 (bit), 436

在缓冲区里 (in buffer), 565

在文件里 (in file), 565

定义 (definition), 71

class, 201

namespace成员声明和 (member declaration and),
150~151

template的上下文 (context of template), 751

常量的, 在类内 (of constant, in-class), 222

定义点 (point of), 753

函数 (function), 129

使用指令和 (using-directive and), 153

虚函数的 (of virtual function), 276

与C声明和定义的差异 (difference from C declaration
and), 714~717.

在类内 (in-class), 210

定址的单位 (addressing, unit of), 80

动态 (dynamic)

存储 (memory), 115

存储 (store), 30

类型检查 (type checking), 638

- 类型检查、误用 (type checking, misuse of), 392
 - 动作 (action), 680
 - 逗号和下标 (comma and subscripting), 733
 - 读 (read)
 - 通过迭代器 (through iterator), 486
 - 行 (line), 544
 - 独立概念 (independent concept), 292
 - 度量、生产率 (measurement, productivity), 629
 - 短 (short) namespace名 (name), 159
 - 短路求值 (short-circuit evaluation), 110
 - 断言检查 (assertion checking), 658
 - 堆 (heap), 31, 346
 - 存储 (memory), 737
 - 存储 (store), 115
 - 和priority_queue (and priority_queue), 480
 - 堆, priority_queue和 (heap, priority_queue and), 424
 - 堆栈 (stack)
 - 存储 (memory), 738
 - 运算符 (operator), 400
 - 对locale敏感的信息 (locale-sensitive message), 813
 - 对返回类型放松 (relaxation of return type), 377
 - 对偶 (pair), 427
 - 对齐 (alignment), 92
 - 对数时间 (logarithmic time), 412
 - 对象 (object), 76
 - I/O, 677
 - static, 222
 - union成员 (member), 218
 - 层次结构 (hierarchy), 656
 - 成员 (member), 218
 - 创建 (creation), 216
 - 创建, 局部性 (creation, localization of), 287
 - 大小可变 (variably-sized), 217
 - 的放置 (placement of), 228
 - 的生存期间 (lifetime of), 76
 - 的种类 (kind of), 218
 - 的状态 (state of), 656
 - 格式 (format), 558
 - 函数 (function), 681
 - 临时 (temporary), 227
 - 全局 (global), 563
 - 数组元素 (array element), 218
 - 真实世界 (real-world), 642
 - 自动 (automatic), 218
 - 自动存储对象的构造函数 (constructor for free store), 220
 - 自由存储 (free store), 218
 - 对象创建的局部化 (localization of object creation), 287
 - 对象的数组 (objects, array of), 224
 - 对修改的响应 (response to change), 613
 - 对元素的要求 (requirements for element), 413
 - 多层次错误处理 (multilevel error handling), 339
 - 多继承 (multiple-inheritance)
 - 和访问控制 (access control and), 360
 - 歧义性消解 (ambiguity resolution), 348
 - 多态的 (polymorphic), 32
 - 对象, 算法和 (object, algorithm and), 56
 - 多态性 (polymorphism), 141
 - dynamic_cast和, 363
 - 编译时间 (compile-time), 310
 - 参数的 (parametric), 310
 - 容器和 (container and), 460
 - 算法和 (algorithm and), 460
 - 运行时 (run-time), 328
 - 见虚函数
 - 多维 (multidimensional)
 - 数组 (array), 733, 735
 - 数组, 传递 (array, passing), 733
 - 多元素 (multiple-element) insert(), 836
 - 多重 (multiple)
 - 继承 (inheritance), 273, 309, 312, 349
 - 继承, 异常和 (inheritance, exception and), 347
 - 继承的使用 (inheritance, use of), 681
 - 界面 (interface), 154
 - 实例化 (instantiation), 758
 - 使用多重继承 (inheritance, using), 355
 - 多重方法 (multi-method), 291
 - 多重继承的使用 (using multiple inheritance), 355
 - 多字节字符编码 (multibyte character encoding), 807
- ## E
- 二分检索 (binary search), 477
 - 二进制模式, binary (binary mode, binary), 562
 - 二进制位 (bit)
 - reference到, 501
 - 定位 (position), 66
 - 模式 (pattern), 436
 - 向量 (vector), 436
 - 域 (field), 112
 - 域 · 735
 - 域, bitset和, 501
 - 二联符 (digraph)
 - %, 726
 - %%, 726

>, 726

:>, 726

<%, 726

<:, 726

二元运算符, 用户定义的 (binary operator, user defined), 235

F

发音 (pronunciation), C++, 9

翻译 (translation), 175

反变 (contravariance), 373

反常、构造函数和析构函数 (anomaly, constructor and destructor), 219

反馈 (feedback), 610

反向迭代器 (reverse iterator), 491

反斜线 \ (backslash), 726

返回 (return)

\r, 回车 (carriage), 726

virtual的返回类型 (type of virtual), 377

函数值 (function value), 253

类型, 放松 (type, relaxation of), 377

类型, 算法 (value, algorithm), 449

类型, 协变 (type, covariant), 377

类型和重载 (type and overload), 135

通过引用 (by reference), 253

值类型检查 (value type check), 132~133

值类型转换 (value type conversion), 132~133

范围 (range)

string的检查 (check of string), 515

检查 (check), 396

检查 (check), valarray, 584

检查 (checking), 684

检查 (checking) Vec, 360

日期 (date), 791

范型, 程序设计 (paradigm, programming), 19

方便的字符分类 (convenient character classification), 806

方便性 (convenience)

和正交性 (and orthogonality), 382

与安全性 (vs, safety), 741

方法 (method), 276

烹调手册 (cookbook), 608

设计 (design), 609

形式化 (formal), 624

选择一种分析 (choosing an analysis), 611

选择一种设计 (choosing a design), 611

方向 (direction)

seekg() 的

seekp() 的

查找的, seekdir (of seek, seekdir)

防火墙 (firewall), 339

访问 (access), 248

不检查的 (unchecked), 396

对 facet, 770

对成员 (to member), 741

对成员类 (to member class), 744

对基类 (to base), 743

检查的 (checked), 396

控制 (control), 201

控制, 强制和 (control, cast and), 366

控制, 使用声明和 (control, using-declaration and), 361

控制, 运行时 (control, run-time), 687

控制和多重继承 (control and multiple-inheritance), 360~361

控制和基类 (control and base class), 360

元素 (element), 395

运算符, 的设计 (operator, design of), 263

放入 (put)

<<, 535

区域 (area), 568

放置 (placement)

new, 228

new(), 508

对象的 (of object), 228

非C++程序 (non-C++ program), 194

非const串文字量, 贬斥 (non-const string literal, deprecated), 716

非标准库 (non-standard library), 40

非成员运算符, 成员和 (nonmember operators, member and), 237

非法迭代器 (invalid iterator), 486

非格式化输入 (unformatted input), 544

非局部 (nonlocal), 318

非类型template参数 (non-type template parameter), 296

非同步事件 (asynchronous event), 319

非修改性序列算法 (nonmodifying sequence algorithm), 463

肥大界面 (fat interface), 390

废料 (garbage)

收集, delete和 (collection, delete and), 738

收集, 析构函数和 (collection, destructor and), 740

收集, 自动 (collection, automatic), 738

收集器 (collector), 117

分别 (separate)

编译 (compilation), 24, 175~176

编译 (compilation), template, 312

分布 (distribution)

- 均匀 (uniform), 602
- 指数 (exponential), 602
- 分隔字符 (separator character), thousands_sep(), 780
- 分号 (semicolon);, 91, 104
- 分解, 功能 (decomposition, functional), 637
- 分类 (classification), 618
 - 方便的字符 (convenient character), 806
 - 字符集 (character), 803
- 分类学 (taxonomy), 618
- 分配 (allocate, allocation)
 - C-风格 (C-style), 509
 - 的单位 (unit of), 100
 - 和释放 (and deallocation), 115
 - 静态 (static), 737
- 分配器 (allocator), 500
 - nothrow, 720
 - Pool_allocator, 505
 - 的复制 (copy of), 820
 - 的使用 (use of), 502
 - 通用 (general), 506
 - 用户定义的 (user defined), 503
- 分配数组 (allocate array), 115
- 分析 (analysis)
 - 错误 (error), 624
 - 方法, 选择一种 (method, choosing an), 611
 - 阶段 (stage), 612
 - 设计和 (design and), 611
 - 试验和 (experimentation and), 623
- 分析, 递归下降 (parser, recursive decent), 97
- 风格, 程序设计 (style, programming), 20
- 封装 (encapsulation), 661
 - 完全 (complete), 253
- 浮点 (floating-point)
 - , 67
 - 类型 (type), 67
 - 输出 (output), 553
 - 提升 (promotion), 729
 - 文字量 (literal), 67
 - 转换 (conversion), 730
 - 转换到 (conversion to), 730
- 符号扩展 (sign extension), 727
- 复杂性, 分治法 (complexity divide and conquer), 609
- 复制 (copy), 204
 - istream, 536
 - ostream, 536
 - 按成员 (memberwise), 253
 - 分配器的 (of allocator), 820

- 赋值 (assignment), 219
- 赋值和template (assignment and template), 310~311
- 构造函数 (constructor), 219
- 构造函数, 默认 (constructor, default), 241
- 构造函数和template (constructor and template), 310~311
- 清除 (elimination of), 594
- 生成的 (generated), 253
- 推迟的 (delayed), 259
- 需求 (requirement), 413
- 异常的 (of exception), 323
- 赋值 (assignment)
 - istream的 (of istream), 536
 - map, 428
 - ostream的 (of ostream), 536
 - string, 518
 - valarray, 582
 - 复制 (copy), 219
 - 函数调用和 (function call and), 89
 - 和template, 复制 (and template, copy), 310~311
 - 和初始化 (and initialization), 253
 - 继承和 (inheritance and), 273
 - 类对象的 (of class object), 219
 - 派生的 (derived), 273
 - 数组 (array), 83
 - 运算符 (operator), 101, 439
 - 自我 (to self), 219
- 覆盖 (override), 351
 - private基类 (base), 647
 - 来自 (from) virtual基类 (base class), 356
- 覆盖函数的类型 (overriding function, type of), 377
- 覆盖和插入 (overwriting vs insertion), 490

G

- 改变 (change), 615
 - 响应 (response to), 613
 - 序列的大小 (size of sequence), 468
 - 增量 (incremental), 600
 - locale, 768
 - 界面 (interface), 677
- 概念 (concept), 13
 - class和 (class and), 199
 - locale, 759
 - 独立 (independent), 292
 - 和类 (and class), 268
- 概念, 类和 (concepts, classes and), 642

- 高级语言 (high-level language), 7
- 高阶函数 (higher-order function), 458
- 格, class (lattice, class), 346
- 格式 (format)
 - %c, 574
 - %d, 574
 - %e, 574
 - %f, 574
 - %G, 574
 - %g, 574
 - %i, 574
 - %n, 574
 - %o, 574
 - %p, 574
 - %s, 574
 - %u, 574
 - %X, 574
 - %x, 574
 - %x, 日期 (date), 793
 - %x, 时间 (time), 793
- 对象 (object), 555
- 货币量的 (of monetary amount), 836~837
- 控制 (control), 550
- 日期的 (of date), 572
- 数 (number), 779
- 信息, locale (information, locale), 534
- 修饰符 (modifier), POSIX, 793
- 整数的 (of integer), 572
- 状态 (state), 550
- 状态, basic_ios (state, basic_ios), 534
- 状态, ios_base (state, ios_base), 534
- 字符 % (character %), 574
- 字符串 (string), 574
- 格式化 (formatting)
 - basic_ios::read, 534
 - 在内存 (in core), 564
- 格式化输出 (formatted output), 550
- 个人 (individual), 629
- 更专门的 (specialized, more), 306
- 工厂 (factory), 288
- 工程, 视图 (engineering, viewgraph), 619
- 工具, 设计 (tools, design), 624
- 公共 (common)
 - 代码和构造函数 (code and constructor), 550
- 公共的 (universal)
 - 基类 (base class), 391
 - 字符名字 (character name), 727
- 公用类成员 (public class member), 201~202
- 功能 (utilities), 383
- 功能, 标准库 (facilities, standard library), 58, 379~380
- 功能的 (functional)
 - 分解 (decomposition), 636
 - 分解与C++ (decomposition and C++), 637
- 共性 (commonality), 268
- 构造 (construction)
 - valarray, 582
 - 部分 (partial), 326
 - 和析构 (and destruction), 218
 - 和析构, 顺序 (and destruction, order of), 367
 - 顺序 (order of), 221
- 构造, 两阶段 (construct, two-stage), 827
- 构造函数 (constructor), 202
 - bitset, 436
 - explicit, 254
 - init() 和 (init() and), 830
 - locale, 765
 - map, 428
 - protected, 769
 - string, 516
 - vector, 396~397
 - virtual, 288
 - 复制 (copy), 253
 - 公共代码和 (common code and), 550
 - 和C风格初始化 (and C-style initialization), 240~241
 - 和template, 复制 (and template, copy), 310~311
 - 和virtual基类 (and virtual base), 353
 - 和不变式 (and invariant), 827
 - 和初始式列表 (and initializer list), 241
 - 和类型转换 (and type conversion), 240
 - 和联合 (and union), 229
 - 和析构函数 (and destructor), 216
 - 和析构函数反常 (and destructor anomaly), 219
 - 和转换 (and conversion), 242
 - 和资源申请 (and resource acquisition), 827
 - 继承和 (inheritance and), 273
 - 默认的 (default), 217
 - 默认的复制 (default copy), 241
 - 生成的 (generated), 253
 - 为局部变量的 (for local variable), 219
 - 为类成员的 (for class member), 220
 - 为内部类型的 (for built-in type), 118
 - 为派生类的 (for derived class), 272~273
 - 为全局变量的 (for global variable), 225

- 为数组元素的 (for array element), 224
- 为自由存储对象的 (for free store object), 220
- 异常和 (exception and), 330
- 指针 (pointer to), 377
- 关闭 (closing)
 - 流 (of stream), 562
 - 文件 (of file), 561
- 关键码 (key)
 - 和值 (and value), 425
 - 惟一 (unique), 425
 - 重复 (duplicate), 433
- 关键字 (keyword), 726
 - and, 726
 - and_eq, 726
 - bitand, 726
 - bitor, 726
 - compl, 726
 - not, 726
 - not_eq, 726
 - or, 726
 - or_eq, 726
 - xor, 726
 - xor_eq, 726
 - 与设计之间的隔阂 (gap between design and), 636
- 关联 (associative)
 - 容器 (container), 425
 - 容器, 序列和 (container, sequence and), 407
 - 容器和异常 (container and exception), 831
 - 数组 (array), 255, 425
 - 数组 见map
- 关系, 编入程序里 (relationship, programmed-in), 654
- 管理 (management), 626
 - 存储 (memory), 737
- 惯性, 组织的 (inertia, organizational), 626
- 广义的 (generalized)
 - 切割 (slice), 595
 - 数值算法 (numeric algorithm), 599
- 归约 (reduce), 600
- 归约 (reduction), 600
- 规模 (scale), 608
 - 的问题 (problems of), 628
- 规模 (scaling), 584
- 国际化, 的途径 (internationalization, approaches to), 760
- 国际货币符号 (international currency symbol), 786
- 过程, 开发 (process, development), 611
- 过程性的 (procedural)
 - 程序设计 (programming), 20

程序设计和C++ (programming and C++), 636

H

函数 (function)

- const成员 (member), 205
- friend, 249
- get(), 665
- init(), 202
- inline, 129
- inline成员 (member), 210
- set(), 665
- static成员 (member), 203
- template, 298
- template参数 (argument), 299
- template重载 (overloading), 300
- virtual, 32, 284
- virtual template, 310~311
- virtual函数的定义 (definition of virtual), 276
- virtual函数的例子 (example of virtual), 568~569
- virtual函数的实现 (implementation of virtual), 31~32
- virtual输出 (virtual output), 539
- 参数传递 (argument passing), 253
- 参数类型, virtual (argument types, virtual), 276
- 参数类型检查 (argument type check), 130
- 参数类型转换 (argument type conversion), 130
- 成员 (member), 212
- 纯virtual (pure virtual), 278
- 调用 (call), 128
- 调用, 与C的差异 (call, difference from C), 713~714
- 调用和赋值 (call and assignment), 89
- 定义 (definition), 129
- 定义, 老风格 (definition, old-style), 713~714
- 定义, 与C的差异 (definition, difference from C), 713~714
- 对象 (object), 76, 216
- 对象, 算术 (object, arithmetic), 458
- 返回值 (value return), 253
- 高阶 (higher-order), 458
- 和算法, C风格 (and algorithm, C-style), 462
- 和异常, C (and exception, C), 840
- 类 (class), 679
- 名字, 重载 (name, overloaded), 133
- 前推 (forwarding), 683
- 嵌套的 (nested), 706
- 声明 (declaration), 128
- 适配器, 指针指向 (adapter, pointer to), 461

探查函数 (inspector) `const`, 620
 体, 以try块为 (body, try-block as), 332
 协助 (helper), 244
 虚 (virtual), 14
 异常和 (exception and), 334
 运算符 :: 和virtual (operator :: and virtual), 278
 值return (value return), 132
 只实例化所用的 (only, instantiate used), 757
 指向 (pointer to), 139
 指向成员函数的指针 (pointer to member), 371
 重载的类型 (type of overriding), 377
 专门化 (specialization), 307
 自立 (free-standing), 643
 函子 (functor), 455
 耗尽 (exhaustion)
 存储器 (memory), 841
 资源 (resource), 328
 自由存储 (free store), 114
 合法的 (valid)
 迭代器 (iterator), 486
 状态 (state), 816
 合作, 设计 (collaboration, design), 622
 和dynamic_cast, 342
 核对 (collating)
 顺序 (order), 778
 序列 (sequence), 302
 黑板作为设计工具 (blackboard as design tool), 618
 宏 (macro), 711
 template和, 753
 替代方式 (alternative to), 144
 与C的差异 (difference from C), 715
 后条件 (postcondition), 660
 后续抽象 (late abstraction), 388
 后缀 (suffix)
 copy, 472
 if, 464
 代码 (code), 550
 划分 (divide)
 分治法, 复杂性 (and conquer, complexity), 609
 划分 (partition), 479
 缓冲 (buffering), 565
 basic_streambuf, 534
 I/O, 567
 缓冲区 (buffer)
 ostream和, 564
 存储 (memory), 507
 里的定位 (position in), 565

唤醒 (resumption), 329
 换行符 (newline) `\n`, 726
 换页符 `\f` (formfeed `\f`), 726
 谎言 (lying), 619
 恢复, 错误 (recovery, error), 842
 回报 (reward), 626
 回车 `\r` (carriage return), 726
 回调, 流 (callback, stream), 572
 回退 (roll-back), 325, 842
 汇编 (assembler), 10
 汇编 (assembler) `asm`, 706
 混成设计 (hybrid design), 630
 混合C和C++ (mixing C and C++), 631
 混合模式算术 (mixed-mode arithmetic), 239
 混入类 (mixin), 357
 货币 (currency)
 符号, 本地 (symbol, local), 786
 符号, 标准 (symbol, standard), 786
 符号, 国际 (symbol, international), 786
 货币 (monetary)
 量的标点 (amount, punctuation of), 785
 量的格式 (amount, format of), 836~837
 量的输出 (amount, output of), 788
 量的输入 (amount, input of), 788

J

机器, 语言和人 (machines, language people and), 8
 机器字 (word), 69
 积 (product)
 点积 (dot), 600
 内积 (inner), 600
 基 (base)
 class, 公共的 (universal), 391
 private, 360
 protected, 360
 成员或 (member or), 649
 访问 (access to), 744
 覆盖private (override private), 647
 和派生类 (and derived class), 35, 646
 类 (class), 269
 类, protected, 651
 类, virtual, 352
 类, 访问控制和 (class, access control and), 360
 类, 私用 (class, private), 651
 类, 由virtual覆盖 (class, override from virtual), 356
 类, 重复的 (replicated), 350
 类的初始化 (class, initialization of), 273

- 类的私用成员 (class, private member of), 272
- 基本保证 (basic guarantee), 816
- 基本运算符 (essential operators), 253
- 基础 (fundamental)
 - 类型 (type), 21, 64
 - 序列 (sequence), 416
- 基础操作 (foundation operator), 620
- 集成 (integration), 639
- 集合 (set), 111
 - 类操作的 (of class operations), 211
 - 在序列上的操作 (operation on sequence), 479
 - 计时器 (timer)
 - 时钟和 (clock and), 789
 - 细粒度 (fine-grained), 789
- 计时一个程序 (timing a program), 789
- 计数, 引用 (counting, reference), 685
- 计算, 数值 (computation, numerical), 56
- 计算器实例 (calculator), 96
- 记法的值 (notation, value of), 233
- 技术 (technique)
 - 内部特征和 (built-in feature vs), 39
 - 语言, 程序设计风格 (language, programming styles), 6
- 继承 (inheritance), 273, 309, 312, 349
 - 包容和 (containment and), 649
 - 多重 (multiple), 154
 - 和 =, 273
 - 和 template, 312
 - 和赋值 (assignment), 273
 - 和构造函数 (constructor), 273
 - 和设计 (design), 621
 - 界面 (interface), 651
 - 模板和 (template and), 309
 - 实现 (implementation), 651
 - 使用多重 (using multiple), 355
 - 使用声明和 (using-declaration and), 349
 - 使用指令和 (using-directive and), 350
 - 数据抽象和 (data abstraction vs), 638
 - 依赖性 (dependency), 646
 - 钻石形 (diamond-shaped), 355
- 加运算符 (plus operator) +, 21
- 间接 (indirection), 257
- 兼容性, C和C++ (compatibility, C and C++), 12, 713~714
- 减量 (decrement)
 - operator--, 112
 - 增量和减量 (increment and), 259
- 减小vector的容量 (decrease capacity of vector), 405
- 减运算符 (minus operator) -, 21
- 检查 (checking)
 - 不变式 (invariant), 656
 - 断言 (assertion), 658
 - 范围 (range), 245
 - 缺少 (missing), 720
 - 异常描述的 (of exception-specification), 335
- 检查, 范围 (check, range), 396
- 检查的 (checked)
 - 迭代器 (iterator), 495
 - 访问 (access), 396
 - 指针 (pointer), 259
- 检索, 二分 (search, binary), 482
- 简单不变式 (simple invariant), 827
- 简单性, 不变式和 (simplicity, invariant and), 827
- 交叉强制 (cross cast), 362
- 交流 (communication), 630
- 教学和C++ (teaching and C++), 12
- 教学作为设计工具 (tutorial as design tool), 622
- 阶段 (stage)
 - 分析 (analysis), 612
 - 开发 (development), 612
 - 设计 (design), 612
 - 实现 (implementation), 612
- 接在文件之后, app (append to file, app), 562
- 节约空间 (saving space), 734
- 结点 (node)
 - class, 抽象 (abstract), 677
 - class, 具体 (concrete), 677
 - 类 (class), 676
- 结构 (structure), 91
 - 程序的 (if program), 7
 - 的初始化 (initialization of), 92
 - 的大小 (size of), 92
 - 内部的 (internal), 609
 - 上的操作 (operations on), 92
- 结果 (result)
 - sizeof的, 109
 - 类型 (type), 109
- 结组, 异常 (grouping, exception), 318
- 截断 (truncation), 730
- 截断文件 (truncate file), trunc, 562
- 解决方案的推广 (solution, generality of), 615
- 解析, 多重继承 (resolution, multiple-inheritance), 346~347
- 界面 (interface)
 - public和protected, 568
 - 层次结构 (hierarchy), 622
 - 多重 (multiple), 154

- 肥大 (fat), 390
- 和实现 (and implementation), 282
- 继承 (inheritance), 354
- 类 (class), 681
- 描述 (specifying), 621
- 模块和 (module and), 148
- 实现和 (implementation and), 282
- 修改 (changing), 677
- 选择 (alternative), 155
- 异常和 (exception and), 334
- 禁止 (prohibiting)
 - &, 236
 - ., 236
 - .., 236
- 经典C locale (classic C locale), 764
- 静态 (static)
 - 存储 (memory), 738
 - 分配 (allocation), 738
 - 类型检查 (type checking), 638
- 局部 (local)
 - static, 129
 - 变量, 构造函数 (variable, constructor for), 219
 - 当前符号 (currency symbol), 786
 - 静态存储 (static store), 224
 - 修正 (fix), 612
 - 作用域 (scope), 75
- 局部化 (locality), 189
- 句柄 (handle)
 - 类 (class), 684
 - 侵入式 (intrusive), 685
- 具体 (concrete)
 - class, 210, 215, 671
 - 结点class (node class), 677
 - 类, 派生自 (class, derive from), 681~684
 - 类型 (type), 29
 - 类型, 抽象和 (type, abstract and), 674~675
 - 类型, 有关问题 (type, problems with), 33
 - 类型, 重用 (type, reuse of), 215
 - 类型和派生 (type and derivation), 672
- 聚集 (aggregate)
 - struct, 91
 - 数组 (array), 91
- 决策, 延迟 (decision, delaying), 620
- 均匀分布 (uniform distribution), 602
- 开发 (development)
 - 过程 (process), 611
 - 阶段 (stage), 612
 - 软件 (software), 608
 - 循环 (cycle), 613
- 开关 (switch)
 - 基于类型 (on type), 370
 - 末次 (last-time), 563
 - 首次 (first-time), 563
- 开始, 程序 (start, program), 191
- 开销 (overhead), 7
- 可靠性 (reliability), 341
- 可扩充的I/O (extensible I/O), 533
- 可扩充性 (extensibility), 615
- 可移植性 (portability), 713
 - 和特征 (and features), 713
- 克隆 (clone), 376
- 空 (null)
 - 0, 零 (zero), 80
 - 指针0 (pointer 0), 730
- 空白 (whitespace), 540~541
 - isspace(), 103
- 空串 (empty string), 516
- 空间, 节约 (space, saving), 734
- 控制, 格式化 (control, format), 550
- 控制语句 (controlled statement), 121
- 库 (library), 40, 128, 563
 - C标准 (C standard), 528
 - 标准 (standard), 40
 - 标准 (standard) 见标准库
 - 初始化 (initialization), 563
 - 第一个C++ (first C++), 603
 - 非标准 (non-standard), 40
 - 功能, 标准 (facilities, standard), 58, 379~380
 - 和异常, 标准 (and exception, standard), 838
 - 容器, 标准 (container, standard), 49, 392
 - 算法, 标准 (algorithms, standard), 56
 - 异常规则 (exception rules for), 841
 - 语言和 (language and), 40
- 库, 标准 (libraries, standard), 615
- 库的规则, 异常 (rules for library, exception), 841
- 宽 (wide)
 - 字符I/O (character I/O), 536
 - 字符分类 (character classification), 530
- 宽字符文字量 (wide-character literal) L', 66
- 框架, 应用 (framework, application), 688
- 扩充的类型信息 (extended type information), 369

括号的使用 (parentheses, uses of), 110

括号的使用 (uses of parentheses), 110

L

老风格函数定义 (old-style function definition), 713~714

类 (class)

 :, 派生 (:, derived), 269

 basic_filebuf, 571

 保护基类 (protected base), 651

 操作, 集合 (operations, set of), 211

 层次结构 (hierarchy), 280, 309

 层次结构, 重组 (hierarchy, reorganization), 621

 层次结构的设计 (hierarchy design), 280

 层次结构和抽象类 (hierarchy and abstract class), 289

 层次结构和模板 (hierarchy and template), 308

 成员, 公用 (member, public), 201

 成员, 构造函数 (member, constructor for), 220

 成员, 私用 (member, private), 201

 抽象 (abstract), 278

 从虚基类覆盖 (override from virtual base), 356

 存储 (storage), 218

 的使用 (use of), 636

 对象, 赋值 (object, assignment of), 219

 概念和 (concept and), 268

 函数 (function), 680

 和类型 (and type), 635

 和设计, 抽象 (and design, abstract), 283~284

 基 (base), 269

 基类的初始化 (initialization of base), 272~273

 基类的私用成员 (private member of base), 272

 结点 (node), 676

 界面 (interface), 681

 句柄 (handle), 684

 派生的 (derived), 273

 派生类的成员 (member of derived), 271

 派生类的构造函数 (constructor for derived), 272~273

 派生类的析构函数 (destructor for derived), 272~273

 私用基类 (private base), 651

 向前引用 (forward reference), 248

 友元 (friend), 249

 指针 (pointer to), 270

 转换到类的指针 (conversion of pointer to), 270

类 (classes)

 的使用 (use of), 643

 和抽象 (and abstraction), 643

 和概念 (and concepts), 642

 和真实世界 (and real-world), 644

 流 (stream), 560

 设计和 (design and), 642

 找出 (finding the), 616

类比 (analogy)

 插接 (plug), 639

 单元 (unit), 639

 汽车工厂 (car factory), 613

 桥梁 (bridge), 635

 通过证明 (proof by), 608

类别 (category)

 迭代器 (iterator), 489

 刻面 (facet), 774~775

类别, locale (category, locale), 762

类别, 消息 (catalog, message), 810

类层次结构的重组 (reorganization of class hierarchy), 621

类型 (type), 21, 64

 char, 字符 (character), 65

 signed, 66

 unsigned, 66

 virtual 的, 返回 (of virtual, return), 377

 安全的 (safe) I/O, 535

 抽象 (abstract), 674

 抽象和具体 (abstract and concrete), 674~675

 等价 (equivalence), 94

 返回的放松 (relaxation of return), 377

 浮点的 (floating-point), 67

 覆盖函数的 (of overriding function), 377

 基本 (fundamental), 21, 64

 基于类型的开关 (switch on), 370

 检查, 动态 (checking, dynamic), 639

 检查, 动态检查的误用 (checking, misuse of dynamic), 392

 检查, 返回值 (check, return value), 132~133

 检查, 函数参数 (check, function argument), 130

 检查, 静态 (checking, static), 638

 结果 (result), 109

 具体 (concrete), 671~672

 具体类型的重用 (reuse of), 673

 类和 (class and), 635

 模块和类型 (module and), 27

 内部 (built-in), 64

 内部类型的构造函数 (constructor for built-in), 118

 内部类型的输出 (output of built-in), 536

 内部类型的输入 (input of built-in), 540

 生成器 (generator), 311

 识别, 运行时 (identification, run-time), 361

 算术 (arithmetic), 63

 协变返回类型 (covariant return), 377

- 信息, 扩充 (information, extended), 369
 - 信息, 运行时 (information, run-time), 361
 - 异常的 (of exception), 338
 - 用户定义的 (user-defined), 199
 - 用户定义的 (user-defined), 64
 - 用户定义类型的string (string of user-defined), 514
 - 用户定义类型的输出 (output of user-defined), 539
 - 用户定义类型的输入 (input of user-defined), 546
 - 用户定义类型的文字量 (literal of user-defined), 243
 - 用户定义运算符和内部运算符 (user-defined operator and built-in), 236~237
 - 与具体类型有关的问题 (problems with concrete), 33
 - 域的 (of field), 68
 - 整数 (integer), 63~64, 66
 - 整数 (integral), 64
 - 整数文字量的 (of integer literal), 728
 - 整数文字量的, 对实现的依赖性 (of integer literal, implementation dependency), 728
 - 指针 (pointer), 502
 - 转换, union和 (conversion, unions and), 736
 - 转换, 返回值 (conversion, return value), 132~133
 - 转换, 构造函数和 (conversion, constructor and), 245
 - 转换, 函数参数 (conversion, function argument), 130
 - 转换, 歧义的 (conversion, ambiguous), 246
 - 转换, 显式的 (conversion, explicit), 116, 253~254
 - 转换, 隐式的 (conversion, implicit), 520
 - 转换, 用户定义的 (conversion, user-defined), 251
 - 转换运算符 (conversion operator), 245
 - 字符 (character), 512
 - 类型安全的连接 (type-safe linkage), 274
 - 类型域 (type-field), 274
 - 类型转换 (type conversion), 245~246
 - 连接 (linkage)
 - const和, 177
 - inline和, 177
 - 错误 (error), 177
 - 和namespace, 184
 - 类型安全 (type-safe), 274
 - 内部 (internal), 177
 - 外部 (external), 177
 - 与C, 182
 - 与指向函数的指针 (and pointer to function), 184
 - 连接器 (linker), 176
 - 联合与类型转换 (unions and type conversion), 737
 - 联系, 效率和 (coupling, efficiency and), 673
 - 两个的规则 (rule of two), 650
 - 两个的规则 (two, rule of), 650
 - 两阶段构造 (two-stage construct), 827
 - 临时量 (temporary), 89
 - 变量 (variable), 227
 - 对象 (object), 227
 - 删除 (elimination of), 594
 - 生存时间 (lifetime of), 113
 - 灵感 (inspiration), 644
 - 灵活性 (flexibility), 615
 - 灵巧指针 (smart pointer), 259
 - 零, 空, 0 (zero null, 0), 80
 - 流 (stream), 384
 - 层次结构 (hierarchy), 560
 - 的关闭 (closing of), 562
 - 迭代器 (iterator), 492
 - 和异常 (and exception), I/O, 839
 - 回调 (callback), 572
 - 类 (classes), 560
 - 文件和 (file and), 560
 - 状态 (state), 542
 - 状态 (state), basic_ios, 534
 - 流失 (leak)
 - 存储 (memory), 818, 843
 - 资源 (resource), 818, 843
 - 逻辑的 (logical)
 - const, 物理的和 (const, physical and), 206
 - 程序的逻辑结构 (structure of program), 176
 - 或运算符 || (or operator ||), 110
 - 与运算符&& (and operator &&), 110
 - 运算符, 按位 (operators, bitwise), 111
- ## M
- 麻烦, struct, 708
 - 枚举 (enumeration), 69
 - 开关 (switch on), 70
 - 枚举符 (enumerator), 69
 - 作为类内常量 (as in-class constant), 222
 - 面向对象 (object-oriented)
 - 程序设计 (programming), 268
 - 纯 (pure), 642
 - 设计 (design), 637
 - 描述界面 (specifying interface), 621
 - 名字 (name), 73
 - locale, 762~764
 - namespace 限定 (qualified), 151
 - 查找 (lookup), namespace, 159
 - 长 (long) namespace, 160
 - 冲突 (clash), 157

- 串 (string), locale, 765
- 短 (short) namespace, 159
- 里的字符 (character in), 73
- 屏蔽 (hiding), 75
- 依赖 (dependent), 750
- 约束 (binding), 751
- 约束 (binding), template, 751
- 名字, 保留的 (names, reserved), 73
- 名字空间 (namespace)
 - 嵌套的 (nested), 741
 - 转变到 (transition to), 164
- 命令行参数 (command line argument), 105
- 命名规则, 迭代器 (naming convention, iterator), 452
- 模板间的关系 (relationships between templates), 311~312
- 模板间的关系 (templates, relationships between), 311
- 模块 (module)
 - 和界面 (and interface), 148
 - 和类型 (and type), 27
- 模块化 (modularity), 280
 - 缺乏 (lack of), 276
- 模块化程序设计 (modular programming), 23
- 模拟 (simulation), 624
 - 事件驱动的 (event driven), 291
- 模式 (pattern), 623
 - 专门化 (specialization), 306
- 模型 (model), 622
 - 瀑布 (waterfall), 612
 - 数学 (mathematical), 624
 - 形式化 (formal), 641
- 末次开关 (last-time switch), 563
- 默认 (default)
 - locale, 768
 - template 参数 (argument), 721
 - 比较 (comparison), 414
 - 参数 (argument), 137
 - 参数值, 例子 (argument value, example of), 202~203
 - 初始式 (initializer), 75
 - 分配器 (allocator), 501
 - 复制构造函数 (copy constructor), 241
 - 构造函数 (constructor), 217
 - 值 (value), 213
 - 值, 提供 (value, supplying), 442
- 目标 (aims)
 - 设计 (design), 615
 - 与手段 (and means), 609
- 目标清晰 (goal, clear), 613
- 目的 (purpose)

- namespace 的, 162
- 程序设计的 (of programming), 609

N

- 内部 (internal)
 - 结构 (structure), 609
 - 连接 (linkage), 177
- 内部 (build-in)
 - 类型 (type), 63
 - 类型, 的构造函数 (type, constructor for), 117
 - 类型, 的输出 (type, output of), 536
 - 类型, 的输入 (type, input of), 540
 - 类型, 用户定义运算符和 (type, user defined operator and), 236
 - 特征与技术 (feature vs technique), 39
- 内积 (inner product), 600
- 拟函数的 class (function-like class), 455
- 匿名联合 (anonymous union), 736

P

- 排错 (debugging), 201
- 排列 (permutation), 482
- 排序 (sort), 482
 - 部分 (partial), 477
 - 稳定 (stable), 477
- 排序 (sorting), 302
 - 准则 (criteria), 472
- 排序序列 (sorted sequence), 476
- 派生 (derive)
 - 从具体类 (from concrete class), 677
 - 没有 virtual (without virtual), 681~684
- 派生, 具体类型和 (derivation, concrete type and), 672~673
- 派生的 (derived)
 - class, 基类和 (class, base and), 646
 - 赋值 (assignment), 273
 - 和友元 (and friend), 745
 - 类 (class), 14
 - 类: (class:), 269
 - 派生类的成员 (class, member of), 271
 - 为派生类的构造函数 (class, constructor for), 272~273
 - 为派生类的析构函数 (class, destructor for), 272~273
 - 异常 (exception), 320
- 庞大化 (gargantuanism), 626
- 烹调手册方法 (cookbook method), 608
- 偏爱的 (preferred) locale, 764

偏移量, 指向成员的指针 (offset, pointer to member and), 373
 拼接, string (concatenation, string), 522~523
 平方时间 (quadratic time), 412
 平衡 (balance), 611
 平衡, 保证和 (tradeoff, guarantee and), 834
 普通算术转换 (usual arithmetic conversions), 109
 瀑布模型 (waterfall model), 612

Q

其他方式

错误处理 (error handling), 172
 对宏 (to macro), 144
 返回 (return), 316
 界面 (interface), 155
 设计 (design), 623
 实现 (implementation), 284

歧义的 (ambiguous)

日期 (date), 793

类型转换 (type conversion), 245~246

歧义性 (ambiguity)

dynamic_cast和, 365

解析, 多重继承 (resolution, multiple-inheritance),
 346~347

汽车工厂类比(car factory analogy), 613

前端操作 (front operation), 418

前条件 (precondition), 660

前推函数 (forwarding function), 683

前向 (forward)

成员类的声明 (declaration of member class), 261

迭代器 (iterator), 485

对类的引用 (reference to class), 248

和输出迭代器 (and output iterator), 486

前缀代码 (prefix code), 549

嵌入 (nesting), 663

嵌套的 (nested)

class, 263

函数 (function), 706

名字空间 (namespace), 741

强保证 (strong guarantee), 817

强制 (coercion), 237

强制去掉const (casting away const), 367

强制转换 (cast)

C风格 (C-Style), 117

贬斥C风格 (deprecated C-style), 716

和访问控制, 366

交叉 (cross), 362

向上 (up), 362

向下 (down), 362

桥梁类比 (bridge analogy), 635

切割 (slicing), 273

切割, 广义的 (slice, generalized), 595

侵入式 (intrusive)

句柄 (handle), 685

容器 (container), 391

消除 (elimination)

程序员 (of programmers), 641

副本 (of copy), 593

临时量 (of temporary), 593

清理, 初始化和 (cleanup, initialization and), 323

清晰的目标 (clear goal), 613

求值 (evaluation)

短路 (short-circuit), 110

顺序 (order of), 110

延迟 (lazy), 621

取出 (get)

定位 (position), tellp(), 565

区域 (area), 568

自 (from), >>, 535

取模运算符 (modulus operator) %, 21

全局 (global), 15

locale, C和C++, 806

namespace, 741

变量 (variable), 203

变量, 构造函数 (variable, constructor for), 224~225

变量的使用 (variable, use of), 99

初始化 (initialization of), 192

对象 (object), 563

作用域 (scope), 741

缺乏模块性 (lack of modularity), 276

缺少 (missing)

bad_alloc, 720

标准库 (standard library), 719

部分专门化 (specialization partial), 720

成员 (member) template, 720

检查 (checking), 720

名字空间 (namespace), 719

R

人的活动, 程序设计是一种 (human activity, programming
 as a), 609

人和机器, 语言 (people and machines, language), 8

日期 (date)

范围 (range), 791

格式 (format of), 571~572

格式%x (format %x), 793
 歧义性 (ambiguous), 800
 输出 (output of), 792
 输入 (input of), 794
 容错 (fault tolerance), 339
 容量 (capacity)
 vector的, 减小(decrease), 405
 vector的, 增大(increase), 405
 容器 (container), 36
 Simula风格 (Simula-style), 391
 Smalltalk风格 (Smalltalk-style), 391
 STL, 391
 string作为 (string as), 435
 valarray作为 (valarray as), 435
 保证 (guarantee), 830
 标准库 (standard library), 50
 存储管理 (memory management), 404
 的表示 (representation of), 413
 的操作 (operation on), 412
 的实现 (implementation of), 413
 迭代器 (iterator), 412
 关联 (associative), 425
 和迭代器 (and iterator), 393
 和多态性 (and polymorphism), 614
 和算法 (and algorithm), 449
 和异常 (and exception), 830, 832
 加入 (adding), 490
 侵入式 (intrusive), 391
 设计 (design), 391
 适配器 (adapter), 416
 输入到(input into), 400
 数组作为 (array as), 439
 序列和 (sequence and), 453
 用户定义 (user-defined), 439~440
 优化 (optimal), 386
 有基的 (based), 391
 种类 (kind of), 407
 综述 (summary), 412
 容器列表 (containers, list of), 383
 冗余 (redundancy), 625
 软件 (software)
 开发 (development), 608
 维护 (maintenance), 625

S

三联符 (trigraphs), 725
 三元运算符 (ternary operator), 560

散列 (hash)
 表 (table), 440
 函数 (function), 445
 函数 (function), hash_map, 440
 散列 (hashing), 445
 莎士比亚 (Shakespeare), 623
 删除 (delete)
 从hash_map, 443
 从序列删除元素 (element from sequence), 472
 商 (quotient), 581
 上下文 (context)
 template定义的 (of template definition), 751
 template实例化的 (of template instantiation), 751
 设计 (design), 611
 []的, 263
 C++ 的, 7, 9
 I/O, 533
 String, 511
 template在设计中, 663~664
 标准库 (standard library), 380~381
 步骤 (steps), 615
 参考文献 (bibliography), 631
 抽象类和 (abstract class and), 283~284
 错误 (error), 624
 的单位 (unit of), 661
 的稳定性 (stability of), 622
 方法 (method), 609
 方法, 选择一种 (method, choosing a), 611
 访问运算符的 (of access operator), 263
 工具 (tools), 624
 工具, 黑板 (tool, blackboard as), 618
 工具, 讲解 (tool, presentation as), 617
 工具, 教学 (tool, tutorial as), 622
 合作 (collaboration), 622
 和程序设计 (and programming), 608
 和分析 (and analysis), 611
 和类 (and classes), 642
 和语言 (and language), 635
 和语言间的隔阂 (and language, gap between), 636
 混成 (hybrid), 630
 继承和 (inheritance and), 621
 阶段 (stage), 612
 类层次结构 (class hierarchy), 280
 面向对象 (object-oriented), 608
 目标 (aims), 615
 容器 (container), 391
 算法 (algorithm), 450

- 完整性 (integrity of), 629
- 为测试 (for testing), 625
- 选择 (alternative), 623
- 语言和程序设计语言 (language and programming language), 641
- 怎样开始 (how to start a), 622
- 重用 (reuse), 623
- 准则, locale (criteria, locale), 582
- 设计的完整性 (integrity of design), 629
- 设计的稳定性 (stability of design), 622
- 设计师和编码员 (designers, coders and), 610
- 设计与语言间的隔阂 (gap between design and language), 636
- 设置 (setting) locale, 768
- 设置位置 (set position), seekp(), 768
- 申请 (acquisition)
 - 构造函数和资源 (constructor and resource), 828, 843
 - 延迟的资源 (delayed resource), 830
 - 资源 (resource), 324
- 申请即初始化 (acquisition is initialization), 325
 - 耗尽 (exhaustion), 329
 - 流失 (leak), 841
 - 释放 (release), 324
- 生产率度量 (productivity measurement), 629
- 生成的 (generated)
 - !=, 415
 - <=, 415
 - , 254
 - >, 415
 - >=, 415
 - 复制 (copy), 254
 - 构造函数 (constructor), 254
 - 专门化 (specialization), 751
- 生成器 (generator)
 - 类型 (type), 311
 - 随机数 (random number), 475
- 生存时间 (lifetime)
 - facet的, 770
 - locale的, 781
 - 对象的 (of object), 76
 - 临时对象的 (of temporary), 227
- 声明 (declaration), 21
 - class, 201
 - friend, 249
 - 成员类的, 前向 (of member class, forward), 261
 - 函数 (function), 128
 - 和定义, namespace成员 (and definition, namespace member), 150~151
 - 和定义, 与C的差异 (and definition, difference from C), 714~717.
 - 声明点 (point of), 75
 - 在for语句里 (in for statement), 122
 - 在条件里 (in condition), 121
 - 声明, 保持一致性 (declarations, keeping consistent), 178
 - 声明运算符 (declarator operator), 72
 - 省略号... (ellipsis ...), 138
 - 失败 (failure), 622
 - 输出 (output), 782
 - 十进制 (decimal), 66
 - 输出 (output), 552
 - 十六进制 (hexadecimal), 66
 - 输出 (output), 555
 - (实际) 参数 (argument)
 - template, 295
 - 变量数 (variable number of), 138
 - 传递, 函数 (passing, function), 129, 253
 - 函数template (function template), 299
 - 类型, 虚函数 (type, virtual function), 276
 - 类型, 与C不同 (type, difference from C), 713~714
 - 类型检查, 函数 (type checking, function), 128
 - 类型转换, 函数 (type conversion, function), 128
 - 命令行 (command line), 105
 - 默认 (default), 137
 - 数组 (array), 131
 - 推断template (deducing template), 299
 - 未声明的 (undeclared), 138
 - 显式template (explicit template), 299
 - 依赖于template (dependency on template), 752
 - 引用 (reference), 88
 - 值, 默认的实例 (value, example of default), 202~203
- 时间 (time)
 - 表示 (representation) time_t, 789
 - 表示 (representation) tm, 789
 - 常量 (constant), 412
 - 的输出 (output of), 792
 - 的输入 (input of), 794
 - 对数 (logarithmic), 412
 - 格式 %X (format %X), 793
 - 平方 (quadratic), 412
 - 线性 (linear), 412
- 时钟和计时器 (clock and timer), 789
- 实例 (example)
 - Assoc, 255
 - Bomb, 818
 - Buffer, 648

cache, 207
 call_from_C(), 342
 callC(), 342
 Car, 676
 Checked, 499
 Checked_iter, 495
 Clock, 354
 Cowboy, 681
 Cvt_to_upper, 809
 Date, 796
 Date_format, 797
 Date_in, 799
 delete_ptr(), 470
 do_it(), 680
 draw_all(), 460
 eliminate_duplicates(), 472
 Employee, 269
 Expr, 377
 Extract_officers, 463
 Filter, 688
 Form, 559
 Hello.world!, 40
 identity(), 470
 Io_obj, 678
 iocopy(), 543
 iosbase::Init, 562
 iseq(), 454
 Ival_box, 361
 Lock_ptr, 326
 math_container, 309
 Matrix, 252
 Money, 784
 My_messages, 810
 My_money_io, 787
 My-punct, 779
 Object, 371
 oseq(), 491
 Plane, 640
 Pool, 503
 Range, 684
 Rational, 655
 Saab, 639
 Safe, 817
 Scrollbar, 652
 Season, 771
 Season <<, 771, 812
 Season >>, 771, 812

Set, 674
 Set_controller, 687
 Shape, 678
 Slice_iter, 589
 sort(), 298
 Stack, 24
 Storable, 351
 String, 293
 String_numput, 781
 Substring, 526
 Table, 217
 Tiny, 245
 Unsafe, 817
 update(), 834
 Vector, 683
 Vehicle, 645
 Window, 354
 成员模板 (member template), 312
 计算器 (calculator), 96
 默认模板参数 (of default argument value), 292
 输入 (of input), 102
 引用 (of reference), 263
 用户定义存储管理 (of user-defined memory management), 346
 云层 (cloud), 615
 运算符重载 (of operator overloading), 266
 (糟糕的), Shape, 368
 实例和重点 (examples and emphasis), 4
 实例化 (instantiation)
 template, 751
 template的上下文 (context of template), 751
 点 (point of), 753
 多重 (multiple), 758
 显式template (explicit template), 757
 直接, template (directive, template), 757
 实现 (implementation)
 dynamic_cast的, 363
 I/O的, 534
 priority_queue, 423
 RTTI的, 362
 Stack, 421~422
 virtual 函数 (function), 31
 标准前的 (pre-standard), 717~718
 迭代器 (iterator), 52
 和界面 (and interface), 282
 继承 (inheritance), 651
 阶段 (stage), 612

- 界面和 (interface and), 282
- 其他方式 (alternative), 285
- 容器的 (of container), 413
- 整数文字量的类型依赖于 (dependency type of integer literal), 728
- 实现定义的 (implementation-defined), 724
- 使用 (use)
 - C++的, 12
 - dynamic_cast的, 678
 - facet的, 770
 - map的, 678
 - rebind的, 502
 - RTTI的, 370
 - template的, 679
 - 多重继承的 (of multiple inheritance), 681
 - 分配器的使用 (of allocator), 502
 - 计数 (count), 293
 - 类的使用 (of class), 636
 - 类的使用 (of classes), 643
 - 全局变量的 (of global variable), 99
 - 依赖性 (dependency), 653
 - 用例 (case), 619
 - 专门化的 (of specialization), 756
- 使用声明 (using-declaration), 151
 - 和访问控制 (and access control), 361
 - 和继承 (and inheritance), 349
 - 与使用指令 (vs. using-directive), 741
- 使用声明与 (using-declaration vs.), 741
- 使用指令 (using-directive), 153
 - 翻译和 (transition and), 165
 - 和定义 (and definition), 162
 - 和继承 (and inheritance), 349
- 事件 (event)
 - 非同步 (asynchronous), 319
 - 驱动模拟 (driven simulation), 291
- 视觉图形工程 (viewgraph engineering), 619
- 试验和分析 (experimentation and analysis), 623
- 是一个 (is-a), 650
- 适配器 (adaptor)
 - 成员函数 (member function), 460
 - 函数指针 (pointer to function), 461
 - 容器 (container), 416
 - 序列 (sequence), 416
- 释放, 分配和 (deallocation, allocation and), 115
- 释放数组 (deallocate array), 115
- 释放资源 (release, resource), 324
- 收集器 (collector)
 - 保守式 (conservative), 740
 - 复制式 (copying), 740
- 手段, 目的和 (means, aims and), 609
- 手工重载解析 (manual overload resolution), 135~136
- 首次开关 (first-time switch), 563
- 受限的字符集 (restricted character set), 725
- 输出 (output), 42
 - <<, 41
 - bitset, 438
 - bool, 537
 - char, 537
 - complex, 598
 - cout, 41
 - C输入和 (input and), 573
 - double, 574
 - float, 574
 - int的位数 (bits of int), 438
 - string, 527
 - valarray, 587
 - 八进制 (octal), 552
 - 操控符 (manipulator), 555
 - 的刷新 (flushing of), 555
 - 迭代器 (iterator), 486
 - 非缓冲的 (unbuffered), 565
 - 浮点数 (floating-point), 553
 - 格式化 (formatted), 550
 - 函数 (function), virtual, 539
 - 货币值的 (of monetary amount), 788
 - 内部类型的 (of built-in type), 536
 - 日期的 (of date), 792
 - 失败 (failure), 782
 - 十进制 (decimal), 550
 - 十六进制 (hexadecimal), 552
 - 时间的 (of time), 792
 - 输入和 (input and), 533
 - 输入和输出间的联系 (connection between input and), 548
 - 数值的 (of numeric value), 780
 - 填充 (padding), 550
 - 为什么用<< (why, <<for), 535
 - 文件的 (to file), 560
 - 序列 (sequence), 491
 - 用户定义类型的 (of user-defined type), 539
 - 域 (field), 554~555
 - 运算符 (operator) <<, 535
 - 整数 (integer), 552
 - 指针的 (of pointer), 538

- 输出与输入的联系 (connection between input and output), 548
- 输入 (input)
 - bitset, 437
 - bool的, 540
 - char的, 544
 - cin >>, 44, 100~101
 - complex, 598
 - string, 527
 - valarray, 587
 - width()的, 542
 - 操控符 (manipulator), 556
 - 从文件 (from file), 560
 - 迭代器 (iterator), 485
 - 非格式化的 (unformatted), 544
 - 非缓冲的 (unbuffered), 565
 - 和输出 (and output), 384
 - 和输出, C (and output, C), 573
 - 和输出间的联系 (and output, connection between), 548
 - 货币量的 (of monetary amount), 788
 - 进入vector, 400
 - 进入容器 (into container), 400
 - 内部类型的 (of built-in type), 540
 - 日期的 (of date), 794
 - 时间的 (of time), 794
 - 实例, 102
 - 数值的 (of numeric value), 783
 - 序列 (sequence), 454
 - 用户定义类型的 (of user-defined type), 546
 - 指针的 (of pointer), 540
- 树 (tree), 274
- 数 (number)
 - 标点 (punctuation), 779
 - 的大小 (size of), 68
 - 格式 (format), 779
- 数据 (data)
 - 成员, 指针指向 (membr, pointer to), 746
 - 抽象 (abstraction), 26
 - 抽象和继承 (abstraction vs inheritance), 638
 - 每个对象 (per-object), 506
 - 每个类型 (per-type), 506
- 数学的 (mathematical)
 - 函数 (functions), complex, 598
 - 函数 (functions), valarray, 586
 - 函数 (functions), vector, 586
 - 函数, 标准 (functions, standard), 580
- 数值 (numeric)
 - 数组 (array), 582
 - 算法, 通用 (algorithm, generalized), 599
 - 限制 (limits), 579
 - 值的输出 (value, output of), 780
 - 值的输入 (value, input of), 783
- 数值计算 (numerical computation), 56
- 数组 (allocate array), 115
- 数组 (array), 80~81, 223
 - new和 (new and), 375
 - string和 (string and), 519
 - valarray和 (valarray and), 582
 - valarray和vector和 (valarray and vector and), 582
 - 布局 (layout), 588
 - 参数 (argument), 131
 - 初始式 (initializer), 81
 - 初始式, 与C的差异 (initializer, difference from C), 714~717
 - 传递多维 (passing multidimensional), 733
 - 的数组 (array of), 732
 - 对象的 (of objects), 224
 - 多维 (multidimensional), 733, 735
 - 分配 (allocate), 115
 - 赋值 (assignment), 83
 - 关联 (associative), 255, 425
 - 聚集 (aggregate), 91
 - 其上的算法 (algorithm on), 482
 - 释放 (deallocate), 115
 - 数值 (numeric), 582
 - 数组的 (of array), 732
 - 异常和 (exception and), 837
 - 用串初始化 (by string, initialization), 81
 - 元素的构造函数 (element, constructor for), 224
 - 元素对象 (element object), 218
 - 指针和 (pointer and), 82, 203
 - 作为容器 (as container), 439
- 刷新输出 (flushing of output), 555
- 双 (double)
 - 引号 (quote), 726
 - 指派 (dispatch), 291
- 双端队列deque (double-ended queue deque), 420
- 双向迭代器 (bidirectional iterator), 484
- 双向链表 (doubly-linked list), 416
- 水平制表符\t (horizontal tab \t), 726
- 顺序 (order), 414
 - 成员初始化 (of member initialization), 221
 - 构造 (of construction), 225
 - 构造和析构 (of construction and destruction), 367

核对 (collating), 778
 求值 (of evaluation), 110
 域的 (of fields), 68
 专门化的 (of specialization), 306
 字符串 (string), 302
 私用 (private)
 基类成员 (member of base class), 272
 类成员 (class member), 201
 算法 (algorithm), 50
 C风格的函数和 (C-style function and), 461~462
 标准库 (standard library), 449
 不修改序列 (nonmodifying sequence), 468
 对数组 (to array), 454
 返回值 (return value), 482
 和string, 515
 和成员函数 (and member function), 460
 和多态对象 (and polymorphic object), 56
 和多态性 (and polymorphism), 614
 和序列 (and sequence), 449
 和异常 (and exception), 839
 容器和 (container and), 449
 设计 (design), 450
 通用数值的 (generalized numeric), 599
 修改序列 (modifying sequence), 37
 异常和 (exception and), 499
 约定 (conventions), 449
 综述 (summary), 449
 算法, 标准库 (algorithms, standard library), 56
 算术 (arithmetic)
 函数对象 (function object), 458
 混合模式 (mixed mode), 239
 类型 (type), 63
 向量 (vector), 57, 582
 运算符 (operator), 21
 指针 (pointer), 68, 79, 85
 转换, 普通 (conversion, usual), 109, 732
 算术if?, 120
 随机 (random)
 数 (number), 476
 数 (number) class, 602
 数 (number) rand(), 602
 数生成器 (number generator), 475
 随机访问迭代器 (random-access iterator), 486
 碎片, 存储 (fragmentation, memory), 740
 缩进编排 (indentation), 123
 索引 (index), 403
 锁定 (locking), 687

T

探查器const函数 (inspector const function), 620
 特殊字符 (special character), 726
 特征 (feature)
 贬斥的 (deprecated), 716
 与技术, 内部 (vs technique, built-in), 39
 总结 (summary), C++716
 特征, 可移植性和 (features, portability and), 713
 特征, 字符 (traits, character), 512
 提供默认值 (supplying default value), 442
 提升 (promotion)
 标准 (standard), 729
 浮点 (floating-point), 729
 整的 (integral), 729
 体系结构 (architecture), 611
 替换 (substitution), Liskov, 652
 添加 (adding)
 facet到localc, 765
 到标准库 (to standard library), 385
 到容器 (to container), 490
 到序列 (to sequence), 490
 填充 (padding), 555
 输出 (output), 550
 条件 (condition), 660
 异常时安全性 (exception safety), 816
 中的声明 (declaration in), 121
 条件表达式 (conditional expression), 120
 跳过初始化, 与C的差异 (jump past initialization, difference from C), 715
 通过迭代器写 (write through iterator), 486
 通过类比证明 (proof by analogy), 608
 通用程序设计语言 (general-purpose programming-language), 19
 通用分配器 (general allocator), 506
 通用型 (generic)
 程序设计 (programming), 663~664
 程序设计, template和 (programming, template and), 292
 算法 (algorithm), 37
 通用性 (generality)
 解决方案的 (of solution), 615
 效率与 (efficiency and), 382
 序列的 (of sequence), 453
 同义词 (synonym) 见typedef
 头文件 (header), 178
 .h, 718

标准库 (standard library), 383
 和编译时间 (and compile time), 186
 文件 (file), 178
 透明性、异常 (transparency, exception), 822
 透视 (vision), 613
 图、有向无环 (graph, directed acyclic), 274
 推迟的 (delayed)
 复制 (copy), 263
 资源申请 (resource acquisition), 830
 推迟决策 (delaying decision), 620
 推断 template 参数 (deducing template argument), 299
 退格 \b (backspace), 726
 椭圆、圆和 (ellipse, circle and), 618

W

外部连接 (external linkage), 177
 完美 (perfection), 38
 完全封装 (complete encapsulation), 253
 完整表达式 (full expression), 227
 唯一密钥 (unique key), 425
 唯一正确道路 (one right way), 609
 惟一定义规则 (one-definition-rule), ODR, 180
 惟一定义规则 ODR (the one-definition-rule), 180
 维护 (maintenance), 180
 软件 (software), 625
 伪装的指针 (disguised pointer), 738
 委托 (delegation), 259
 未捕捉的异常 (uncaught exception), 338
 未初始化的存储 (uninitialized memory), 506
 未声明的参数 (undeclared argument), 138
 未预期的异常 (unexpected exception), 336
 位数、大小 (mantissa, size of), 579
 位组 (bits)
 char 里, 578
 float 里, 578
 int 里, 578
 谓词 (predicate), 457, 838
 标准库 (standard library), 457
 和异常 (and exception), 838
 用户定义 (user-defined), 457
 文档 (documentation), 627~628
 文化习俗, locale (cultural preference, locale), 759
 文件 (file)
 .c, 179
 .h, 178
 打开 (opening of), 561
 关闭 (closing of), 562

和流 (and stream), 560
 里的定位 (position in), 565
 模式 (mode of), 562
 输出 (output to), 561
 输入 (input from), 561
 头文件 (header), 178
 源 (source), 175
 文件的模式 (mode of file), 562
 文字量 (literal)
 ', 字符 (character), 66
 L', 宽字符 (wide-character), 66
 String, 264
 浮点 (floating-point), 67
 用户定义类型的 (of user-defined type), 244
 整数 (integer), 66, 69
 整数文字量的类型 (type of integer), 728
 整数文字量的类型依赖于实现 (implementation dependency type of integer), 728
 字符串 (string), 82
 稳定的 (stable)
 list merge(), 417
 list sort(), 417
 排序 (sort), 476
 问题 (problems)
 规模的 (of scale), 628
 具体类型的 (with concrete type), 33
 无定义的 (undefined)
 enum 转换 (conversion), 70
 行为 (behavior), 724
 行为、异常和 (behavior, exception and), 818
 无缓冲的 (unbuffered)
 I/O, 570
 输出 (output), 565
 输入 (input), 565
 无名的 (unnamed)
 namespace, 158
 union, 736
 物理的 (physical)
 程序的物理结构 (structure of program), 176
 和逻辑 (and logical) const, 206
 误用 (misuse)
 C++, 636
 RTTI, 370
 动态类型检查 (of dynamic type checking), 392

X

析构 (destruction)

- 构造和(construction and), 218
- 构造和析构的顺序 (order of construction and), 367
- 析构函数 (destructor), 30
 - ~和, 217
 - atexit() 和, 194
 - virtual, 284
 - 反常, 构造函数和 (anomaly, constructor and), 219
 - 公共代码和 (common code and), 550
 - 构造函数和 (constructor and), 216
 - 和废料收集 (and garbage collection), 740
 - 和联合 (and union), 229
 - 为派生类 (for derived class), 272~273
 - 显式调用 (explicit call of), 228
 - 异常和 (exception and), 332
 - 异常时安全性 (exception safety and), 816
- 系统 (system)
 - 成长 (growing), 624
 - 成功的大型 (successful large), 623
 - 工作 (working), 623
- 细粒度计时器 (fine-grained timer), 789
- 下标 (subscript)
 - C++风格 (style), 592
 - Fortran风格 (style), 592
- 下标 (subscripting), 403
 - map, 427
 - string的, 516
 - valarray, 582
 - 逗号 and (comma and), 733
 - 用户定义的 (user-defined), 255
- 下溢 (underflow), stack, 422
- 显式的 (explicit)
 - template参数 (template argument), 299
 - template实例化 (template instantiation), 757
 - 调用析构函数 (call of destructor), 228
 - 类型转换 (type conversion), 116, 253~254
 - 限定 :: (qualification ::), 741
- 线性时间 (linear time), 412
- 限定::, 显式 (qualification::, explicit), 741
- 限定词, template作为 (qualifier, template as), 750
- 限定名 (qualified name), namespace, 151
- 限制 (restriction), 9
- 限制, 数值 (limits, numeric), 579
- 相等 (equality)
 - hash_map, 440
 - 和比较 (and comparison), 405
 - 没有== (without==), 415
- 相互引用 (mutual reference), 248
- 想法, 真实世界是其源泉 (ideas, real-world as source of), 644
- 向量 (vector)
 - Fortran, 587
 - 操作 (operations), 584
 - 数学函数 (mathematical functions), 586
 - 算术 (arithmetic), 582
 - 位 (bit), 111
 - 指数 (exponentiation), 586
- 向上强制 (up cast), 362
- 向下强制 (down cast), 362
- 向序列添加元素 (add element to sequence), 468
- 消息 (message)
 - queue, 423
 - 对locale敏感 (locale-sensitive), 813
 - 门类 (catalog), 810
- 小于 (less)
 - 小于等于运算符 <= (than or equal operator <=), 21
 - 小于运算符 < (than operator <), 21
- 效率 (efficiency), 626
 - 操作的 (of operation), 412
 - 和联系 (and coupling), 674
 - 和通用性 (and generality), 382
- 协变返回类型 (covariant return type), 377
- 协助 (helper)
 - class, 263
 - 函数 (function), 244
 - 函数和 (function and) namespace, 123~214
- 写时复制, 263
- 信号 (signal), 318
- 信息隐藏 (information hiding), 24
- 行, 读 (line, read), 544
- 行为, 无定义 (behavior, undefined), 393
- (形式) 参数 (parameter)
 - template, 296
 - 非类型 (non-type) template, 296
- 形式化 (formal)
 - 方法 (method), 624
 - 模型 (model), 641
- 雄心勃勃 (ambition), 608
- 修改 (modify)
 - locale, 768
 - map, 431
- 修改操作 (modifier), 620
 - POSIX格式 (format), 793
- 修改序列的算法 (modifying sequence algorithm), 468
- 虚的 (virtual)
 - 函数 (function), 14

函数, 重新命名 (function, renaming), 682

序列 (sequence), 37, 416

- lexicographical_compare(), 481
- max_element(), 481
- min_element(), 481
- string, 511
- 半开 (half-open), 453
- 从中删除元素 (delete element from), 472
- 错误 (error), 453
- 的普遍性 (generality of), 453
- 迭代器和 (iterator and), 486
- 改变大小 (change size of), 468
- 和范围 (and range), 450
- 和关联容器 (and associative container), 403
- 和容器 (and container), 453
- 基本 (fundamental), 416
- 加入 (adding to), 490
- 加入元素 (add element to), 468
- 排序的 (sorted), 477
- 上的集合操作 (set operation on), 479
- 适配器 (adapter), 416
- 输出 (output), 491
- 输入 (input), 454
- 算法, 非修改 (algorithm, nonmodifying), 463
- 算法, 修改 (algorithm, modifying), 468
- 算法和 (algorithm and), 449

序列, 重叠 (sequences, overlapping), 468

序列的 min_element() (of sequence), 481

选择 (choosing)

- 分析方法 (an analysis method), 611
- 设计方法 (a design method), 611

选择运算 (selecting operations), 619

学习 (learning)

- C++, 717~718
- C++, 逐步方式 (gradual approach to), 7
- C和C++, 7

循环 (loop)

- 合并 (merging), 681
- 和错误 (and error), 463
- 语句 (statement), 104

Y

延迟求值 (lazy evaluation), 621

要求 (requirement)

- 比较 (comparison), 414
- 复制 (copy), 413

叶class (leaf class), 677

一元运算符, 用户定义 (unary operator, user-defined), 235

依赖的名字 (dependent name), 752

依赖性 (dependency), 646, 653, 724, 728

- 参数化和 (parameterization and), 621
- 创建 (create), 654
- 继承 (inheritance), 646
- 使用 (use), 654
- 硬件 (hardware), 724
- 于template参数 (on template argument), 752
- 最小化 (minimize), 671

移位状态 (shift state), 807

遗产 (legacy), 622

以报告作为设计工具 (presentation as design tool), 619

异常 (exception), 48, 122, 168, 320, 334

- atexit() 和, 340
- bad_alloc, 508
- bsearch() 和, 840
- clear() 和, 834
- complex 和, 840
- C函数和, 840
- C和, 341
- deque 和, 832
- erase() 和, 832
- goto 和, 123
- I/O, 547
- I/O流和 (stream and), 839
- insert() 和, 832
- iostream 和, 839
- istream 和, 839
- list 和, 832
- map 和, 832
- new 和, 508
- ostream 和, 839
- pop_back() 和, 832
- pop_front() 和, 832
- push_back() 和, 832
- push_front() 和, 832
- qsort() 和, 840
- remove() 和, 832
- remove_if() 和, 832
- runtime_error, 764
- sort() 和, 832
- splice() 和, 832
- string 和, 839
- swap() 和, 836
- uninitialized_copy() 和, 837, 839

- uninitialized_fill() 和, 837, 839
- uninitialized_fill_n() 和, 837, 839
- unique() 和, 832
- valarray 和, 840
- vector 和, 832
- 安全性 (safety), 815
- 安全性, 等级 (safety, degrees of), 819
- 安全性, 技术 (safety, techniques for), 819
- 安全性和析构函数 (safety and destructor), 817
- 安全性条件 (safety condition), 817
- 保证 (guarantee), 817
- 保证总结 (guarantee summary), 833
- 标准 (standard), 342
- 标准库和 (standard library and), 838
- 捕捉一切 (catch every), 48
- 不变式和 (invariant and), 816
- 初始化和 (initialization and), 837
- 处理 (handling), 842
- 处理器 (handler), 7, 110
- 的辨识 (discrimination of), 168
- 的代价 (cost of), 335
- 的复制 (copy of), 323
- 的类型 (type of), 338
- 迭代器和 (iterator and), 837
- 关联容器和 (associative container and), 832
- 和 <<, 782
- 和 main(), 48
- 和 new, 327
- 和不变式 (and invariant), 827
- 和成员 (and member), 818
- 和成员初始化 (and member initialization), 332
- 和错误 (and error), 547
- 和递归函数 (and recursive function), 333
- 和多重继承 (and multiple inheritance), 347
- 和构造函数 (and constructor), 331
- 和函数 (and function), 333
- 和界面 (and interface), 334
- 和数组 (and array), 838
- 和算法 (and algorithm), 499
- 和无定义行为 (and undefined behavior), 818
- 和析构函数 (and destructor), 332
- 和子对象 (and sub-object), 325
- 结组 (grouping), 318
- 库的规则 (rules for library), 841
- 派生的 (derived), 319
- 容器和 (container and), 815, 832
- 算法和 (algorithm and), 839
- 透明性 (transparency), 822
- 未捕捉的 (uncaught), 338
- 未预期的 (unexpected), 336
- 谓词和 (predicate and), 838
- 引用和 (reference and), 837
- 映射 (mapping), 336
- 指针和 (pointer and), 837
- 异常层次结构 (exception hierarchy), 342~343
- 异常的辨识 (discrimination of exception), 168
- 异常的代价 (cost of exception), 335
- 异常描述 (exception-specification), 334
- 异常描述 (exception-specification)
 - 检查 (checking of), 335
- 异常时安全的代码 (exception-safe code), 822
- 异常时安全性的等级 (degrees of exception safety), 819
- 异或运算符 (exclusive or operator ^, bitwise), 111
- 抑止代码膨胀 (curbing code bloat), 306
- 意义 (meaning)
 - C++, 9
 - 标识符的 (of identifier), 749
 - 运算符, 预定义 (for operator, predefined), 236
- 溢出 (overflow), 堆栈 (stack), 422
- 引号 (quote)
 - \, 单 (single), 726
 - 双 (double), 726
- 引入C++ (introducing C++), 630
- 引用 (reference)
 - &, 88
 - dynamic_cast 到, 363
 - return, 132
 - 捕捉 (catch by), 320
 - 参数 (argument), 89
 - 成员 (member), 649
 - 成员初始化 (member initialization), 224
 - 初始化 (initialization of), 89
 - 到 class, 前向 (forward), 248
 - 调用 (call by), 251
 - 返回 (return by), 253
 - 和异常 (and exception), 837
 - 计数 (count), 293
 - 计数 (counting), 685
 - 实例, 261
 - 相互, (mutual), 248
- 引用上的操作 (references, operations on), 88
- 隐藏 (hiding)
 - 名字 (name), 75
 - 信息 (information), 24

- 隐式的 (implicit)
 - string 转换 (conversion), 520
 - 类型转换 (type conversion), 69, 245~246, 251, 253~254
 - 应用 (application)
 - 框架 (framework), 641
 - 运算符 (operator), 254~255
 - 映射 (map), 425
- 映射到的类型, 值 (mapped type, value), 49
- 映射异常 (mapping exception), 336
- 硬件 (hardware), 69
 - 依赖性 (dependency), 724
- 用户定义的 (user-defined)
 - =, 472
 - facet, 774
 - facet season_io, 771
 - 操控符 (manipulator), 558
 - 存储管理, 实例 (memory management, example of), 346
 - 迭代器 (iterator), 495
 - 二元运算符 (binary operator), 235
 - 分配器 (allocator), 505
 - 类型 (type), 64
 - 类型, class (type, class), 199
 - 类型的 string (type, string of), 514
 - 类型的输出 (type, output of), 539
 - 类型的输入 (type, input of), 546
 - 类型的文字量 (type, literal of), 243
 - 类型转换 (type conversion), 251
 - 容器 (container), 440
 - 谓词 (predicate), 457
 - 下标 (subscripting), 255
 - 一元运算符 (unary operator), 235
 - 运算符 (operator), 235
 - 运算符 (operator) --, 259
 - 运算符 (operator) +, 251
 - 运算符 (operator) ++, 251
 - 运算符 (operator) +=, 251
 - 运算符 (operator) =, 251
 - 运算符 (operator) ->, 257~259
 - 运算符和枚举 (operator and enum), 236~237
 - 运算符和内部类型 (operator and built-in type), 236~237
 - 指针转换 (pointer conversion), 312
 - 转换 (conversion), 310
- 用户提供的比较 (user-supplied comparison), 414
- 用户专门化 (user specialization), 751
- 优化容器 (optimal container), 387
- 优先队列 (priority queue), 423
- 优先级 (precedence)
 - <<, 536
 - 运算符 (operator), 109
- 游戏 (game), 602
- 有关异常时安全性的技术 (techniques for exception safety), 819
- 有基的容器 (based container), 391
- 有向无环图 (directed acyclic graph), 274
- 有一个 (has-a), 650
- 余数 (remainder), 581
- 语法 (grammar), 695
- 语法 (syntax)
 - <, template, 710
 - 总结 (summary), 695
- 语句 (statement)
 - break, 104
 - continue, 104
 - do, 122
 - for, 122
 - goto, 122
 - if, 119
 - switch, 119
 - while, 122
 - 控制 (controlled), 121
 - 循环 (loop), 104
 - 综述 (summary), 118
- 语言 (language)
 - support, 385~386
 - 程序设计 (programming), 14
 - 程序设计风格技术 (programming styles technique), 6
 - 低级 (low-level), 7
 - 高级 (high-level), 7
 - 和库 (and library), 40
 - 人和机器 (people and machines), 8
 - 设计和 (design and), 635
- 语义 (semantics)
 - 值 (value), 262
 - 指针 (pointer), 262
- 预处理指令 (preprocessing directive), #, 711
- 预定义 (predefined)
 - ' , 236
 - &, 236
 - =, 236
 - 运算符的意义 (meaning for operator), 236
- 域 (field)
 - ∴ 位 (bit), 735
 - 类型 (type of), 67

- 输出 (output), 554~555
- 位 (bit), 112
- 域, 顺序 (fields, order of), 68
- 元素 (element)
 - 从序列里, 删除 (from sequence, delete), 468
 - 到序列里, 加入 (to sequence, add), 468
 - 的地址 (address of), 403
 - 第一个 (first), 396
 - 对象, 数组 (object, array), 218
 - 对元素的要求 (requirements for element), 413
 - 访问 (access), 396
 - 访问, list (access, list), 417
 - 访问, map (access, map), 426
 - 数组元素的构造函数 (constructor for array), 224
 - 最后一个 (last), 396
- 元素地址 (address of element), 403
- 原始存储 (raw storage), 502
- 原型 (prototypes), 623
- 圆和椭圆 (circle and ellipse), 618
- 源 (source)
 - 代码 (code), template, 312~313
 - 文件 (file), 175
 - 想法, 真实世界是其源泉 (ideas, real-world as source of), 644
- 约定, 习惯 (conventions)
 - 词法 (lexical), 695
 - 算法 (algorithm), 449
- 约束 (binding)
 - 名字 (name), 751
 - 强度, 运算符 (strength, operator), 107~109
- 云层的实例 (cloud example), 615~616
- 运算符 (operator)
 - `'`, 110
 - `!=`, 不等 (not equal), 21
 - `%`, 取模 (modulus), 21
 - `&&`, 逻辑与 (logical and), 110
 - `&`, 按位与 (bitwise and), 111
 - `*`, 乘 (multiply), 21
 - `-`, 减 (minus), 21
 - `--`, 减量 (decrement), 112
 - `--`, 用户定义 (user-defined), 259~260
 - `.`, 746
 - `.`, 成员访问 (member access), 91
 - `/`, 除 (divide), 21
 - `::`, 272
 - `::`, 作用域解析 (scope resolution), 75, 131
 - `::` 和 virtual 函数 (function), 278
 - `?:`, 120
 - `^`, 按位异或 (bitwise exclusive or), 111
 - `|`, 按位或 (bitwise or), 111
 - `||`, 逻辑或 (logical or), 110
 - `~`, 按位补 (bitwise complement), 111
 - `+`, 加 (plus), 21
 - `+`, 用户定义 (user-defined), 251
 - `++`, 用户定义 (user-defined), 259~260
 - `++`, 增量 (increment), 112
 - `+-`, 98
 - `+-`, 用户定义 (user-defined), 236
 - `<`, 小于 (less than), 21
 - `<<`, 输出 (output), 534
 - `<=`, 小于等于 (less than or equal), 21
 - `--`, 98
 - `=`, 用户定义 (user-defined), 251
 - `=`, 等于 (equal), 21
 - `-> *`, 746
 - `->`, 成员访问 (member access), 92
 - `>`, 大于 (greater than), 21
 - `->`, 用户定义 (user-defined), 257~259
 - `>=`, 大于等于 (greater than or equal), 21
 - delete, 115
 - new, 115
 - 比较运算符 (comparison operator), 21
 - 的结合性 (associativity of), 109
 - 调用 (call), 256
 - 堆栈 (stack), 399
 - 访问运算符的定义 (design of access), 263
 - 赋值 (assignment), 101, 439
 - 和枚举, 用户定义 (and enum, user-defined), 236
 - 和内部类型, 用户定义 (and built-in type, user-defined), 236
 - 基础 (foundation), 620
 - 类型转换 (type conversion), 245
 - 三元 (ternary), 560
 - 声明符 (declarator), 72
 - 算术 (arithmetic), 21
 - 应用 (application), 256
 - 用户定义 (user-defined), 235
 - 用户定义二元 (user-defined binary), 235
 - 用户定义一元 (user-defined unary), 235
 - 优先级 (precedence), 109
 - 预定义意义 (predefined meaning for), 236
 - 约束强度 (binding strength), 535
 - 运算符比较 (operator comparison), 21
 - 重载 (overloaded), 215

- 重载, 实例 (overloading, example of), 346
- 综述 (summary), 107
- 组合 (composite), 239
- 运算符 (operators)
 - n元 (n-ary), 593
 - 按位逻辑 (bitwise logical), 111
 - 成员和非成员 (member and nonmember), 237
 - 和namespace, 237
 - 基本 (essential), 253
- 运算符的结合性 (associativity of operator), 109
- 运算符函数表 (functions, list of operator), 234
- 运行时 (run-time)
 - 初始化 (initialization), 192
 - 错误 (error), 314
 - 多态性 (polymorphism), 310
 - 访问控制 (access control), 687
 - 类型识别 (type identification), 367
 - 类型信息 (type information), 677
 - 支持 (support), 7

Z

- 在类内 (in-class)
 - 常量, 枚举符作为 (constant, enumerator as), 222
 - 常量的定义 (definition of constant), 222
 - 初始式 (initializer), 222
 - 定义 (definition), 210
- 在内存格式化 (in core formatting), 564
- 责任 (responsibility), 620
- 怎样开始设计 (how to start a design), 622
- 增加 (increase)
 - vector的大小 (size of vector), 404~405
 - vector的容量 (capacity of vector), 405
- 增量 (increment)
 - 和减量 (and decrement), 259
 - 运算符 (operator) ++, 112
- 增量变化 (incremental change), 600
- 找出类 (finding the classes), 644
- 真实世界 (real-world)
 - 对象 (object), 642
 - 类和 (classes and), 644
 - 作为灵感的源泉 (as source of ideas), 644
- 诊断 (diagnostics), 384
- 整数 (integer)
 - enum和, 70
 - 的格式 (format of), 571
 - 类型 (type), 63~64, 66

- 类型, 转换到 (type, conversion to), 730
- 输出 (output), 552
- 文字量 (literal), 66, 69
- 文字量, 类型依赖于实现 (literal, implementation dependency type of), 728
- 文字量的类型 (literal, type of), 728
- 转换 (conversion), 730
- 转换 (conversion), signed unsigned, 730
- 整型 (integral)
 - 类型 (type), 63
 - 提升 (promotion), 729
 - 转换到 (conversion to), 730
- 正交性, 方便性和 (orthogonality, convenience and), 382
- 正在工作的系统 (working system), 623
- 支持 (support), 627
 - 运行时 (run-time), 8
- 直接操作 (direct manipulation), 641
- 值 (value)
 - cin的, 246
 - return, 函数 (function), 132~133
 - 捕捉 (catch by), 316
 - 调用 (call by), 130
 - 返回 (return by), 132~133
 - 关键码和 (key and), 425
 - 函数返回的 (return, function), 253
 - 记法的 (of notation), 233
 - 默认的 (default), 213
 - 映射到的 (mapped type), 49
 - 语义 (semantics), 262
 - 字符的 (of character), 512
- 只实例化所用的函数 (used function only, instantiate), 757
- 只实例化所用函数 (instantiate used function only), 757
- 子数组 (subarray), 595~597
- 指令 (directive)
 - # 预处理 (preprocessing), 711
 - template实例化 (instantiation), 757
- 指派, 双 (dispatch, double), 291
- 指数, vector (exponentiation, vector), 586
- 指数的大小 (exponent, size of), 579
- 指数分布 (exponential distribution), 602
- 指针 (pointer), 79
 - 0, 空 (null), 730
 - const, 85
 - 成员或 (member or), 647
 - 到 (to) void, 90
 - 到const, 85

- 到成员 (to member) .*, 372
- 到成员 (to member) ::*, 372
- 到成员函数 (to member function), 372
- 到成员和偏移量 (to member and offset), 373
- 到构造函数 (to constructor), 377
- 到函数 (to function), 139
- 到函数 (to function), <<, 555
- 到函数 (to function), >>, 556
- 到函数 (to member) ->*, 372
- 到函数, 连接与 (to function, linkage and), 184
- 到函数适配器 (to function adapter), 461
- 到类 (to class), 270
- 到类, 的转换 (to class, conversion of), 270
- 到数据成员 (to data member), 746
- 的大小 (size of), 68
- 的输出 (output of), 537
- 的输入 (input of), 540
- 和数组 (and array), 131
- 和异常 (and exception), 837
- 检查的 (checked), 259~260
- 类型 (type), 502
- 灵巧 (smart), 259
- 算术 (arithmetic), 112
- 伪装的 (disguised), 738
- 语义 (semantics), 262
- 转换 (conversion), 730
- 转换, 用户定义 (conversion, user-defined), 312
- 指针和 (pointers and) union, 738
- 制表符 (tab)
 - \c, 水平 (horizontal), 726
 - \v, 垂直 (vertical), 726
- 质量 (quality), 630
- 终止 (termination), 329
 - 程序 (program), 193
- 种类 (kind)
 - 对象的 (of object), 218
 - 类的 (of class), 670
 - 容器的 (of container), 403
- 周期, 开发 (cycle, development), 613
- 逐步 (gradual)
 - 采纳 (adoption of) C++, 630
 - 学习C++的方法 (approach to learning C++), 6
- 逐步成长的系统 (growing system), 624
- 主导 (dominance), 381
- 主动性 (initiative), 610
- 注释 (comment), 123
 - */, 24
 - /*, 144
 - /* 开始 (/* start of), 24
 - //, 9
- 专门化 (specialization), 751
 - template, 303
 - 部分的 (partial), 305
 - 部分的, 缺少 (partial, missing), 720
 - 的使用 (use of), 756
 - 的顺序 (order of), 323
 - 函数 (function), 307
 - 和 char*, 307
 - 和 void*, 303
 - 模式 (pattern), 305
 - 生成的 (generated), 751
 - 用户 (user), 751
- 转变 (transition), 630~631
 - 到名字空间 (to namespace), 163
 - 和使用指令 (and using-directive), 163
- 转换 (conversion), 620
 - complex, 598
 - signed unsigned 整数 (signed unsigned integer), 730
 - string, 519
 - 到 bool (to bool), 730
 - 到浮点数 (to floating-point), 730
 - 到类的指针 (of pointer to class), 270
 - 到整数类型 (to integer type), 730
 - 到整型 (to integral), 730
 - 浮点数 (floating-point), 730
 - 构造函数和 (constructor and), 242
 - 构造函数和类型 (constructor and type), 240
 - 歧义类型 (ambiguous type), 246
 - 无定义的 enum (undefined enum), 70
 - 显式类型 (explicit type), 116
 - 隐式类型 (implicit type), 69, 245~246, 251, 253~254
 - 用户定义 (user-defined), 309~310
 - 用户定义类型 (user-defined type), 250
 - 用户定义指针 (user-defined pointer), 312
 - 运算符, 类型 (operator, type), 245
 - 整数 (integer), 730
 - 指针 (pointer), 730
 - 字符串的, 隐式 (of string, implicit), 520
 - 字符模式 (character code), 807
- 转换 (conversions), 655
 - 普通算术 (usual arithmetic), 109
- 转换字符表示 (converting character representation), 807

- 转义字符 \ (escape character \), 726
- 装载器/装入器(loader), 175
- 状态 (state)
 - 错误 (error), 816
 - 对象的 (of object), 656
 - 格式化 (format), 550
 - 合法 (valid), 816
 - 机器 (machine), 641
 - 流 (stream), 542
- 准则 (criteria)
 - 标准库 (standard library), 381
 - 排序 (sorting), 472
- 资源 (resource)
 - 申请 (acquisition), 324
 - 申请, 构造函数和 (acquisition, constructor and), 842
 - 申请, 推迟 (acquisition, delayed), 830
- 子串 (substring), 526
- 子对象, 异常和 (sub-object, exception and), 818
- 子范围 (subrange), 684
- 子类 (subclass), 270
 - 超类和 (superclass and), 35
- 子类型 (subtype), 651~652
- 字典 (dictionary), 425
- 字典 (dictionary) 见map
- 字符 (character), 512
 - s, 73
 - %, 格式化 (% format), 574
 - \, 转义字符 (\, escape), 66, 726
 - _, 73
 - 16位 (16-bit), 512
 - mask, 803
 - thousands_sep() 分隔符 (thousands_sep() separator), 780
 - 本国 (national), 725
 - 编码, 多字节 (encoding, multibyte), 807
 - 编码转换 (code conversion), 807
 - 表示, 转换 (representation, converting), 807
 - 串 (string), 384
 - 的值 (value of), 512
 - 分类 (classification), 803
 - 分类, 方便 (classification, convenient), 806
 - 分类, 宽 (classification, wide), 530
 - 缓冲区, streambuf和 (buffer, streambuf and), 565
 - 集合 (set), 725
 - 集合, ASCII (set, ASCII), 66, 530
 - 集合, 大 (set, large), 725
 - 集合, 受限的 (set, restricted), 725
 - 类型 (type), 512
 - 类型char (type char), 65
 - 名字, 通用的 (name, universal), 725~728
 - 特殊 (special), 726
 - 特征 (traits), 512
 - 文字量 ' (literal '), 66
 - 在名字里 (in name), 73
- 字符串 (string)
 - locale名字, 765
 - 比较 (comparison), 776
 - 的大小 (size of), 132
 - 格式 (format), 574
 - 和const, C风格, 81
 - 顺序 (order), 778
 - 文字量 (literal), 66, 69, 292
 - 文字量, 贬斥非const (literal, deprecated non-const), 716
 - 用于初始化数组 (initialization of array by), 80~81
 - 用于字符串比较的locale (comparison, locale used for), 768
- 字符 (character), 384
- 字节 (byte), 69
- 自动 (automatic)
 - 存储 (memory), 737
 - 存储管理 (memory management), 738
 - 对象 (object), 218
 - 废料收集 (garbage collection), 220
- 自立 (free-standing)
 - class, 642
 - 函数 (function), 643
- 自然的操作 (natural operation), 672
- 自我赋值 (self, assignment to), 823
- 自引用 (self-reference) this, 205
- 自由 (free)
 - 存储 (store), 30, 115, 218, 507, 737, 800
 - 存储对象 (store object), 218
 - 存储对象, 构造函数 (store object, constructor for), 220
 - 存储耗尽 (store exhaustion), 115
- 总结 (summary)
 - 容器 (container), 412
 - 算法 (algorithm), 449
 - 异常时保证 (exception guarantee), 833
 - 语法 (syntax), 695
- 组合, namespace (composition, namespace), 160
- 组合器 (compositor), 595
- 组合运算符 (composite operator), 239

- 组件 (component), 613
 - 标准 (standard), 613
- 组织 (organization)
 - I/O系统的 (of I/O system), 534
 - 标准库 (standard library), 383
- 组织的惯性 (organizational inertia), 626
- 钻石形继承 (diamond-shaped inheritance), 355
- 最大的 `int` (largest `int`), 578
- 最后元素 (last element), 396
- 最小的 (smallest) `int`, 578
- 最小化 (minimalism), 620
- 最小化依赖性 (minimize dependency), 671
- 最优化 (optimization), 594
- 左值 (lvalue), 251
- 作用域 (scope), 248
 - 标号的 (of label), 122
 - 操纵符和 (manipulator and), 556
 - 和重载 (and overload), 135
 - 解析运算符:: (resolution operator::), 203
 - 局部 (local), 75
 - 全局 (global), 741
 - 与C的差异 (difference from C), 713~714
 - 作用域错误 (domain error), 581